Learning the vi and Vim Editors

第7版
7 New Chapters on Vim



学习 vi 和 Vim 编辑器（中文版）

Arnold Robbins,
Elbert Hannah & Linda Lamb 著
*O'Reilly Taiwan*公司 编译

O'REILLY®
東南大學出版社

O'REILLY®

学习vi和Vim编辑器



将近三十年的时间里，vi一直是Unix与Linux采用的标准编辑器，从1986年开始，本书一直是第一线的vi导引手册。但三十年来，Unix已不再是三十年前的样子，这本书也不能一成不变。第7版的《学习vi和Vim》涵盖了Vim的详细指引，Vim是一种很棒的vi同类品。

Vim现在是大多数Linux系统上的默认编辑器，也是Mac OS X的默认vi版本，同时能在许多其他操作系统上执行。本书将说明使用这两种程序编辑文档的基础技巧，并讨论高级工具，例如交互式宏与扩展编辑器的脚本——我们的内容编写成容易遵循步骤操作的风格，成就本书的经典地位。读者将学到：

- 快速于vi里移动
- 超越vi基础的技巧，例如使用缓冲区
- 使用vi的全局搜索与替换功能
- 自定义vi，并执行Unix的命令
- 使用Vim的扩展文本对象以及威力强大的正则表达式
- 执行多窗口编辑，并设计Vim脚本
- 充分利用图形化用户界面版的Vim，gvim
- 使用Vim的强化功能，例如语法高亮显示及扩展标签
- Vim与其他三种vi同类品的比较：nvi、elvis、vile

vi或Vim，是使用Linux或Unix时的必要知识，无论使用哪种平台，本书都是基础中的基础。

"vi，就像许多Unix早年开发出的经典公共程序，都有难以驾驭的评价。Bram Moolenaar的强化版同类品——Vim走了很长的路才消除这种印象……它大概已经变成最受欢迎的vi版本。"

——前言

Arnold Robbins，专业程序员与技术作家，从1980年即开始使用Unix系统，协助打造了awk的POSIX标准。

Elbert Hannah，是位专业软件工程师与软件设计师，于1983年使用汇编语言写出全屏编辑器，完成第一件受指派的专业任务。

Linda Lamb，O'Reilly Media的第一代编辑中的一员，也是本公司的技术作家与营销经理。

www.oreilly.com

O'Reilly Media, Inc.授权东南大学出版社出版

ISBN 978-7-5641-2604-9



定价：82.00元

学习vi和Vim编辑器

Arnold .Linda Lam Linda Lamb & Linda Lamb 著
Taiwan公司 编译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc.授权东南大学出版社出版

东南大学出版社

图书在版编目（CIP）数据

学习 vi 和 Vim 编辑器 /（美）罗宾斯（Robbins, A.），（美）汉娜（Hannah, E.），（美）拉姆（Lamb, L.）著；O'Reilly Taiwan 公司编译 .—南京：东南大学出版社，2011.3

书名原文：Learning the vi and Vim Editors, 7E

ISBN 978-7-5641-2604-9

I. ①学… II. ①罗… ②汉… ③拉… ④ O… III. ① UNIX 操作系统－文本编辑程序 IV. ① TP316.81

中国版本图书馆 CIP 数据核字（2011）第 004459 号

江苏省版权局著作权合同登记

图字：10-2009-239 号

学习 vi 和 Vim 编辑器

出版发行：东南大学出版社
地　　址：南京四牌楼 2 号　　邮编：210096
出 版 人：江建中
网　　址：http://www.seupress.com
电子邮件：press@seu.edu.cn
印　　刷：扬中市印刷有限公司
开　　本：787 毫米 × 980 毫米　16 开本
印　　张：29.5 印张
字　　数：578 千字
版　　次：2011 年 3 月第 1 版
印　　次：2011 年 3 月第 1 次印刷
书　　号：ISBN 978-7-5641-2604-9
印　　数：1~3000 册
定　　价：82.00 元（册）

本社图书若有印装质量问题，请直接与读者服务部联系。电话（传真）：025-83792328

O’Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O’Reilly Media, Inc.授权东南大学出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O’Reilly Media, Inc.是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的*The Whole Internet User' Guide & Catalog*（被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O’Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O’Reilly Media, Inc.是最稳定的计算机图书出版商 —— 每一本书都一版再版。与大多数计算机图书出版商相比，O’Reilly Media, Inc.具有深厚的计算机专业背景，这使得O’Reilly Media, Inc.形成了一个非常不同于其他出版商的出版方针。O’Reilly Media, Inc.所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O’Reilly Media, Inc.还有许多固定的作者群体 —— 他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O’Reilly Media, Inc.依靠他们及时地推出图书。因为O’Reilly Media, Inc.紧密地与计算机业界联系着，所以O’Reilly Media, Inc.知道市场上真正需要什么图书。

目录

第二部分 Vim

前言

在任何计算机系统中，文本编辑是最常见的任务，而vi是最有用的标准文本编辑器之一。vi可以创建新文件或是编辑既有的纯文本文件。

vi，像许多于Unix早期开发的经典实用工具一样，有一个难于驾驭的名声。Bram Moolenaar的增强同类品，Vim（vi Improved），对于消除产生此印象的原因大有帮助。Vim包含无数的便利、视觉指南以及帮助画面。它或许成为最流行的vi版本，所以本书的第7版在第二部分“Vim”中奉献了7个新的章节给它。然而，也存在许多其他有价值的vi同类品，我们将其中三个涵盖在第三部分“其他vi同类品”中。

本书的范围

这本书共有18章与4篇附录，分成4个部分。第一部分，基础与高级的vi，可以让你对vi很快地上手；接下来是高级的技巧，可以让你的工作更有效率。

前面两章，第一章“vi文本编辑器”与第二章“简单的文本编辑”，介绍了一些简单的vi命令，供初学者上手。你应该多多练习，直到熟悉为止。当你在第二章学到一些基础的编辑工具后，便可以稍事休息。

但是vi可不是只能做基本的文本编辑而已，它的各种命令与选项都可以简化编辑工作。第三章“快速移动位置”与第四章“越过基础的藩篱”，重点都是完成任务的简化方式。第一次阅读的时侯，你只需要大致了解vi可做的事以及哪些命令可能对你特别有用即可。因为以后你可以随时回到这些章节，做更深入的研究。

第五章“ex编辑器概述”、第六章“全局替换”与第七章“高级编辑方法”，提供了一些工具，可以让你把许多繁重的编辑工作交给计算机。其中介绍了位于vi底层的ex行编辑器，并且示范了如何在vi中使用ex命令。

第八章“vi同类品的功能总览”，介绍了本书涵盖的四种vi同类品所具备的扩展功能。着重于探讨多窗口的编辑、图形用户界面（GUI）、扩展的正则表达式（regular expression）等等简化编辑的功能及其他特色，为后续内容提供概略总图。本章还指出原始vi源代码的取得方式，以便在时下的Unix系统（包括GNU/Linux）上轻易地编译vi。

第二部分“Vim”，则说明一种最受欢迎的vi同类品（就21世纪初期而言）。

第九章“Vim（vi Improved）概述”是对Vim的通论，包括何处可取得用于各种常见操作系统的Vim二进制版本以及一些使用Vim的不同方式。

第十章“Vim对vi的主要改进”，如题所述是描述Vim在vi之上的重大改善，例如内置帮助、对初始化的控制、额外的移动命令，还有扩展的正则表达式。

第十一章“Vim的多窗口功能”，重点在于多个窗口的编辑，这或许是标准vi的最重大附加功能。本章提供所有创建与使用多重窗口的细节。

第十二章“Vim脚本”，深入探讨Vim的命令语法，可编写脚本来自定义或修改Vim以符合需求。大部分创造性的Vim易用功能来自其他用户贡献的脚本，让Vim一并发布。

第十三章“图形化Vim（gvim）”，用于查看Vim在时下GUI环境里的发展，例如商用Unix系统上的标准、GNU/Linux与其他类似Unix的产品以及MS Windows。

第十四章“程序员专用的Vim强化功能”，着重在Vim作为程序员的编辑器，如何拥有超越一般的文本编辑能力。折叠（folding）与大纲功能、智慧缩排、语法高亮显示、“编辑—编译—调试”周期的加速是其中特别有价值的功能。

第十五章“其他好用的Vim功能”，这章有点集大成的意味，涵盖了许多有趣但不适合放在稍早章节的重要内容。

第三部分“其他vi同类品”，讲述了另外三种常用的vi同类品：nvi、elvis、vile。

第十六章“nvi：新的vi”、第十七章“Elvis”、第十八章“vile：类似Emacs的vi”，介绍各种vi的同类品——nvi、elvis、vile，除了讲述如何使用它们扩展自vi的功能，也讨论了各自的特色。

第四部分“附录”，提供了一些有用的参考资料。

附录A“vi、ex与Vim编辑器”，列出所有vi与ex命令，以功能排序。同时以字母顺序列出ex命令。另外还收录了用于Vim的精选vi与ex命令。

附录B“设置选项”，列出vi与本书所涵盖的四种同类品所用的set命令选项。

附录C“问题集”，是本书中常见问题的大集合。

附录D“vi与国际互联网”，描述了vi在广大Unix与Internet文化中的地位。

本书写作的方式

撰写这本手册是为了把我们认为vi新手必须了解的知识，让读者有概括性的认识。学习一种新的编辑器并不是容易的事，尤其是像vi的选项这么复杂，更是辛苦。我们努力将基本的概念与命令用浅显易读的方式呈现出来。

在讨论过（到处都适用的）vi的基础后，我们换个话题，深入讨论Vim。然后再回头在vi的范畴内，查看nvi、elvis与vile。接下来说明本书使用的编排惯例。

vi命令的讨论

例如左边的键盘按键图案，标志着这个特殊键盘命令或是相关命令的主要讨论区。你会看到对该命令主要概念的一段简短介绍，接着是介绍各项目的段落。然后会介绍不同任务适合的各种命令，并加上命令的描述与正确的使用语法。

排版约定

在语法的描述与范例中，需要实际键入的内容以`Courier`字体表示，它们都是命令名称。文件名称与程序选项也使用`Courier`字体。变量（不会直接打出来，而是在键入命令时用实际值代替的内容）则是用*Courier*斜体表示。方括号表示这个变量为可选变量。例如，以下这行语法：

```
vi[filename]
```

其中*filename*用实际的文件名来替换。方括号代表命令vi可以不必加上文件名。括号本身不必输入。

某些范例会显示在Unix的shell提示符下输入命令产生的结果。其中，你实际需输入的内



按键

特殊的按键会显示在方框中。例如：

iWith a [ESC]

在本书中，你也会发现一整栏的vi命令及其结果：

按键	结果
ZZ	"practice"[Newfile]6lines,320characters 输入ZZ这个写入与保存命令后，你的文件会存储成一般的Unix文件。

在上面的范例中，ZZ命令在左栏，窗口右边是屏幕上的一行（或多行），显示命令的结果。光标位置由高亮字符表示。本例的ZZ会写入并存储文件，因此会见到状态行，光标则不会显示出来。下方的文字简单地解释了命令与结果。

有时需要同步按下[CTRL]键与其他的键才能使用某个vi命令。在文本中，这种组合键是写在方框里（例如[CTRL-G]）。在范例中，则是在此键的前面加上脱字符号（^）。例如，^G表示按住[CTRL]键同时按下[G]键。

问题集

一些问题的解答分布在章节中。你可以跳过这些问题解答，在以后有需要时再回来翻阅。另外，所有的问题解答都集合在附录C中，以方便查阅。

预备知识

这本书假定你已经读过《Learning the Unix Operating System》（O'Reilly出版）或是其他介绍Unix的书籍。你应该已经知道如何：

- 登录与注销
- 输入Unix命令
- 更换目录
- 列出目录中的文件
- 创建、复制与移除文件

熟悉grep（一种全局搜索程序）和通配符也很有用。

建议与评论

请将对本书的评价和存在的问题通过如下地址告知出版者：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly的每一本书都有专属网站，你可以在那找到关于本书的相关信息，包括勘误表、示例代码以及其他的信息。本书的网站地址是：

http://www.oreilly.com/catalog/9780596529833/

对于本书的评论和技术性的问题，请发送电子邮件到：

Bookquestions@oreilly.com

关于本书的更多信息、会议、资源中心和O'Reilly网络，请访问以下网站：

http://www.oreilly.com/
http://www.oreilly.com.cn/

关于旧版

本书的第五版（当时的书名是《Learning the vi Editor》）首次详细讨论了ex编辑器。在第五、第六、第七章中，ex与vi的复杂功能也用了更多的范例来阐明，像正则表达式的语法、全局替换、.exrc文件、缩写词、键盘映射与编辑脚本等。有些范例是从《Unix World》杂志的文章中节录而来。Walter Zintz写了关于vi的分成两部分的教学手册（注1），指导了一些我们不知道的事，也有许多高技巧性的范例介绍了一些本书谈到的功能。Ray Swartz在他的专栏中也提到一些小技巧（注2）。我们很感谢这些文章中的概念。

注1：“vi Tips for Power Users”，*Unix World*，1990年4月刊；“Using vi to Auto mate Complex Edits”，*Unix World*，1990年5月刊。作者均为Walter Zintz。（附录D中有这两篇文章的网址。）

注2：“Answers to Unix”，*Unix World*，1990年8月刊。

第六版的《Learning the vi Editor》涵盖四种免费、可自由取得的vi同类品或类似vi的编辑器。它们多半都改进了原始的vi。现在可以说已经有vi编辑器的“系列”出现了，而本书的目的，便是说明使用它们所需的知识。这个版本对nvi、Vim、elvis、vile一视同仁。

第六版中还新增了如下内容：

- 文中许多细节的更正并新增了文字。
- 在适当的章节最后加上命令的摘要表。
- 新的章节包含了各种vi同类品，介绍了它们之间共同的特性与扩展功能以及多窗口编辑。
- 介绍各种vi同类品的章节中，描述了一些同类品的历史与目标、独特性以及从何处可以取得。
- 新的附录描述了vi在广大Unix与Internet文化中的地位。

关于第7版的前言

《学习vi和Vim编辑器》（第7版）延续了第六版的所有优秀特色。时间证明Vim是最受欢迎的vi同类品，所以本版特别加重介绍了这个编辑器的内容（也出现在了书名里）。不过，为了尽量照顾广大的vi同类品用户，我们也继续更新了nvi、elvis、vile的内容。

新增内容

以下是本版新增的内容：

- 更正了基本内容的错误。
- 以7个章节详尽地、无遗漏地介绍Vim。
- 更新了nvi、elvis、vile的素材。
- 前一版中关于ex与vi的两份参考附录合并为一章，并包含了精选的Vim相关内容。
- 其他附录也已更新。

版本

我们测试vi的特性时是使用以下程序：

- 以Solaris版的vi作为Unix vi的参考版本
- Keith Bostic的nvi 1.79版

- Steve Kirkendall的elvis 2.2版
- Bram Moolenaar的Vim7.1版
- Kevin Buettner、Tom Dickey与Paul Fox的vile 9.6版

第六版的致谢

首先，最感谢我的妻子Miriam，在我忙着编写这本书时照顾小孩，尤其是饭前的“重要时刻”（witching hours）。我要补偿她很多安静休息的时间与冰淇淋。

乔治亚理工学院计算机学院的Paul Manno在安抚我的打印软件上，提供了难以报答的协助。O'Reilly & Associates的Len Muellner与Eric Ray则提供了SGML软件方面的协助。Jerry Peek的vi宏对SGML也是无价之宝。

虽然在准备本书新内容与整理旧内容的期间已使用过所有提到的程序，但大多数编辑工作仍是使用Vim 4.5与5.0完成，操作平台则是GNU/Linux（Red Hat 4.2）。

感谢Keith Bostic、Steve Kirkendall、Bram Moolenaar、Paul Fox、Tom Dickey、Kevin Buettner帮我们审阅内容。Steve Kirkendall、Bram Moolenaar、Paul Fox、Tom Dickey、Kevin Buettner也对第八章到十二章贡献良多（本处是指第六版的章节编号）。

若没有电力公司供电，根本不可能使用计算机完成任何工作；但是当电力正常供应时，我们根本不会想到缺乏它的景象。写书也是如此——没有编辑，根本不能完成工作；但当编辑执行任务时，又很容易忘记他们的存在。O'Reilly的Gigi Estabrook真是其中瑰宝，与她合作非常愉快，我很感谢她为我付出的一切。

最后，对O'Reilly & Associates的出版团队致上莫大谢意。

——ArnoldRobbins

Ra'anana,ISRAEL

June1998

第7版的致谢

Arnold再次感谢妻子Miriam的关怀与支持。欠她的安静的个人时间与冰淇淋持续累积中。另外，感谢J.D.“Illiad”Frazer提供的最棒的*User Friendly*系列漫画（注3）。

Elbert想谢谢Anna、Cally、Bobby与家中的父母，谢谢他们在艰难的时刻仍对他的工作维持高度兴趣。他们的热情很有感染力，非常感谢。

注3：没听过这系列漫画的人，请参考*http://www.userfriendly.org*。

谢谢Keith Bostic与Steve Kirkendall，在我们重新审视关于他们所做的编辑器的章节时提供了珍贵的意见。Tom Dickey对`vile`与附录B的表格提供了独到见解。Bram Moolenaar（Vim的作者）这次也参与本书的审阅。Robert P.J. Day、Matt Frye、Judith Myerson、Stephen Figgins则对文章内容贡献了重要的评论。

Arnold与Elbert还想谢谢Andy Oram与Isabel Kunkle的编辑工作，还有O'Reilly Media的出版团队与提供的工具。

——ArnoldRobbins

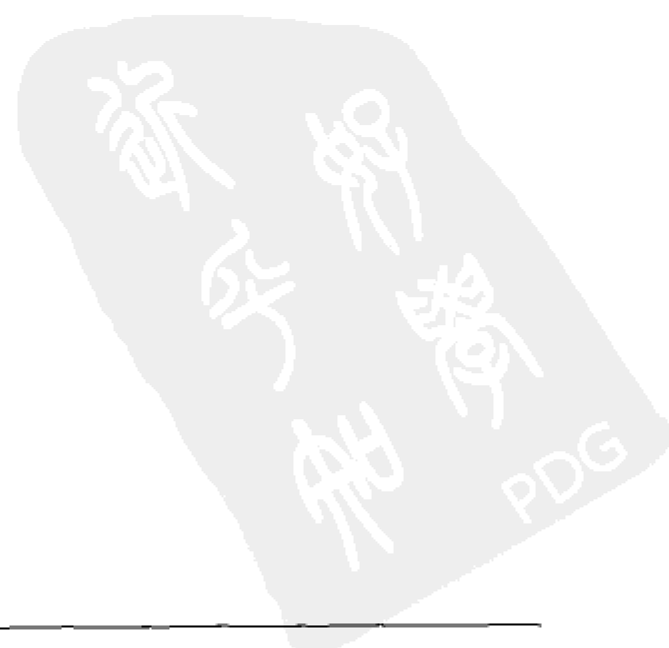
NofAyalon,ISRAEL

2008

——ElbertHannah

Kildeer,IllinoisUSA

2008



第一部分

基础与高级的vi

第一部分是用来让你对vi尽快上手，并提供高级的技巧，让工作更有效率。该部分包含下列章节：

- 第一章，*vi文本编辑器*
- 第二章，*简单的文本编辑*
- 第三章，*快速移动位置*
- 第四章，*越过基础的藩篱*
- 第五章，*ex编辑器概述*
- 第六章，*全局替换*
- 第七章，*高级编辑方法*
- 第八章，*vi同类品的功能总览*



第一章

vi文本编辑器

Unix（注1）中有许许多多的编辑器，可以处理各种文本文件，不管那些文件包含数据、源代码还是语句。其中有些是行编辑器，像ed或ex，一次只在屏幕上显示文件中的一行；有的是全屏编辑器，像vi或emacs，可以在屏幕上显示文件的一部分。在X Window系统下的编辑器也很常见，并且越来越风行。GNU Emacs与衍生出的XEmacs都提供了多个X窗口的功能；另外还有两种有趣的选择，Bell Labs的sam与Acme编辑器。Vim也提供了基于X窗口的界面（X-based interface）。

在你的系统中，vi是最有用的标准文本编辑器（vi是visual editor——可视化编辑器的简写，读作“vee-eye”。请参考图1-1）。它几乎在所有的Unix系统上有近乎相同的形式，因此可以说是一种文本编辑器的“通用语”，这一点跟Emacs很不一样（注2）。当然ed与ex也一样有“通用性”，但是一般来说，还是全屏编辑器比较好用（事实上，行编辑器一般已不被使用）。全屏编辑器可以卷页、移动光标、删除一整行、插入文字等等，同时可以立刻看到修改的结果。全屏编辑器非常风行，因为它可以让你在阅读文件时当场做修改，就像修改印出来的稿件一样，而且速度更快。

对于许多新手来说，vi不但不直观，而且麻烦——它不以特殊控制键用于文字处理并让你能够正常输入，反而使用所有的常用键来下命令。当按键用于下命令时，vi是处于“命令模式”（command mode）。你必须先进入“插入模式”（insert mode），才能输入实际的文字。而且，命令似乎多如牛毛。

注1：最近以来，“Unix”一词同时包含了衍生自原始Unix code base的商用系统以及源代码可用的Unix相似系统。Solaris、AIX、HP-UX是商用系统的代表，GNU/Linux及各种BSD衍生系统则是后者的代表。除非特别注明，本书的一切内容都适用于上述系统。

注2：GNU的Emacs已成为通用的Emacs版本。它的唯一问题就是并未成为大多数商用Unix系统的标准，我们必须自行取得并安装Emacs。

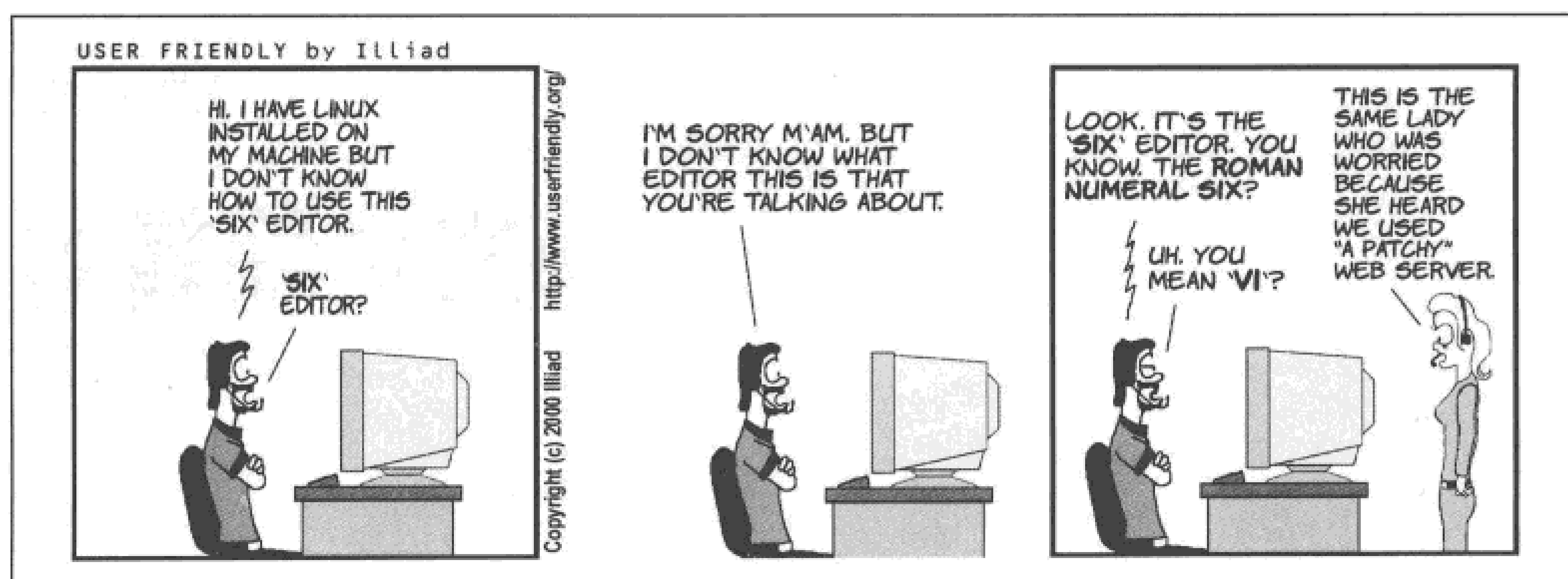


图1-1：vi的正确发音

然而，当你开始学习之后，将了解vi的确经过精心设计。你只需要按几个键，便可以完成复杂的工作。在学习vi的过程中，你也将学着把编辑工作逐渐交给计算机——这原本就是计算机的工作。

vi（像任何文本编辑器一样）并不是一个“所见即所得”的文字处理程序。如果你需要生成格式化的文件，必须自己输入其他格式化程序所需的命令，以便控制打印的结果。以缩排数段文字为例，你需要在缩排开始与结束的地方插入命令。格式化命令可以让你试验或者更动打印出来的文件外观。从许多方面来说，它比文字编辑器提供更多文件外观的控制权。Unix支持troff格式化程序（注3）。而TEX与LATEX也是很常用的格式化程序（注4）。

（vi支持一些简单的格式化机制。例如，你可以要求它在一行结束时环绕，或自动缩排新行。此外，Vim第7版还提供了自动拼写检查。）

你用vi做的文本编辑工作越多，就会变得越简单，能做的事也越多，这和其他技术是一样的。一旦你习惯了vi所拥有的编辑能力，可能不会想再回头使用任何“简易的”编辑器了。

文本编辑工作有哪些？首先，你会想要插入文本（一个忘掉的单词或是句子），接着会删除文本（错字或整个段落）。你也会想要更动字母或词（更改拼写错误或是改变某个用词）。也可能是将文本从文件的一处移动到另一处，或是复制到另一处。

注3：troff用于激光打印机与排字机（typesetter）。它的“孪生兄弟”是nroff，用于行式打印机和终端机。依照Unix 的惯例，我们把两者并称为troff。现在，任何使用troff的人都在使用它的GNU版：groff。可至*http://www.gnu.org/software/groff*进一步了解更多信息。

注4：可分别至*http://www.ctan.org*与*http://www.latex-project.org*了解更多关于TEX与LATEX的信息。

但vi的初始状态与其他文字处理程序不一样，它的“默认”状态是命令模式：只需要几个键，就可以做复杂的、交互式的编辑（要插入文本，只要从“插入”命令中挑一个，就可以开始输入文字了）。

基本的命令可能有一个或两个字符，例如：

i

插入（insert）

cw

更改字词（change word）

用字母作为命令可以大幅增加速度。你不需要死记一大堆的功能键，或是为了按出组合键而相当不自然地伸展手指。大部分命令都可利用相关字母记忆，几乎所有的命令都有类似的模式并且互相关联。

一般来说，vi的命令有如下特点：

- 字母大小写有区别（大写与小写表示不同的意义，I与i功用不同）。
- 在输入时不会显示在屏幕上。
- 不需要在命令后加上Enter键。

同时有另一组命令显示在屏幕的底端，而这些命令前有特殊的符号。斜线（/）与问号（?）用于开始搜索命令，将在第三章讨论。冒号（:）用于开始所有的ex命令。ex命令是ex行编辑器使用的命令，在你使用vi时，也可以利用ex，因为它是底层的编辑器，vi只是其“可视化”的模式而已。ex命令与概念将在第五章讨论，本章只介绍退出文件而不保存的ex命令。

简史

在深入了解vi的命令与操作前，先介绍一点它的历史，这有助于理解操作环境上的vi的世界观，并特别有助于想通许多让人不解的vi错误消息，同时，还将观察到各种vi同类品是如何超越原始的vi的。

vi的历史可回溯到计算机用户还在终端机上操作，必须通过serial line（串行线路）与中央小型计算机连接的时代。连在中央计算机上的终端机可能有好几百台，而且分布在世界各地。每台终端机都能做同样的动作（如清除屏幕、移动光标），但动作所需的命令各不相同。除此之外，Unix系统能让用户选择用于退格（backspace）、产生中断信号及其他适合用于串行终端机（serial terminal）的指令，例如暂缓与继续输出。这些功能均使用stty命令管理（现在也仍然是）。

原始的UCB版的vi是从源代码中抽出有关终端控制的信息（这些信息很难改变），成为终端能力（terminal capability）的文本文件数据库（它比较容易改变），其管理则通过termcap函数库。20世纪80年代初期，System V引入一个二元的终端信息（terminal information）数据库与terminfo函数库。以上两个函数库在功能上大致相同。为了告诉vi采用哪个函数库，你必须设置环境变量TERM。这个变量通常在shell启动文件中设置，例如.profile或.login。

现在，大家都使用图形环境的虚拟终端（例如xterm），系统几乎也都为我们设置好了TERM（当然，你也能在个人计算机的非GUI控制台下使用vi，这在单一用户模式下修复系统时非常好用。不过，现在已经没有太多人愿意把这种方式当成日常工作的一环了）。在日常使用时，你很可能想要GUI版的vi，例如Vim或其他同类产品。在Microsoft Windows或Mac OS X系统上，GUI版的编辑器大都是默认编辑器；然而，在虚拟终端上运行vi（或相同年代的其他全屏编辑器），仍然需使用TERM以及termcap或terminfo，并需注意stty的设置。当然，在虚拟终端上使用它，就跟学习vi的方式一样简单。

关于vi，还有一项重要的事实需要了解。与现在的看法相比，在开发它的时候，Unix系统被视为较不稳定的系统。过去的vi用户必须随时应付系统不定时的死机（crash），所以vi支持恢复正在编辑中的文件（注5）。所以，当各位学习vi，看到各种关于潜在问题的说明时，请记得这些过往的发展。

打开与关闭文件

你可以使用vi编辑任何文本。vi将要编辑的文件复制到缓冲区（内存中另外设置的暂存本地内容的部分），显示缓冲区（虽然一次只能看到一个屏幕尺寸的部分），并且让你增加、删除与更改文本。存储编辑的结果时，vi则把缓冲区中的内容写回到永久的文件中，替换同名的旧文件。有一点要记住，你永远是在缓冲区里的文件副本上作业，除非存储缓冲区，否则编辑结果不会影响原始的文件。存储编辑的结果也称为“写入缓冲区”或是更常见的“写入文件”。

打开文件

vi是调用vi编辑器、编辑新旧文件所用的Unix命令。vi命令的语法是：

```
$ vi [filename]
```

注5：幸好这种事情已经不再是常态了，虽然系统还是会因为外部因素而死机（crash），例如电力中断。

上述命令行中出现了方括号，表示括号中的filename是个可选项，可有可无。方括号本身不用输入。$是Unix提示符。如果省略filename，vi会打开一个未命名的缓冲区。我们可在将缓冲区里的内容写入文件时命名。不过我们还是应保持良好习惯，先在命令行上给出文件名称。

文件名在目录中必须是唯一的。文件名可以包括除了斜线（/）与ASCII NUL以外的任何8位字符：斜线保留作路径名称中文件与目录的分隔之用，而ASCII NUL全部的位都是0。你甚至可以在文件名中包含空格，只要在前面加上反斜线（\）即可。实际上，文件名通常包含任意的大写与小写字母组合，再加上点号（.）与下划线（_）字符等。请记住，Unix区分大小写，小写字母与大写字母视为不同字符，还要记得按下Enter键，告诉Unix你已经结束命令了。

在目录中打开新文件时，应该在vi命令中加上新的文件名。例如要在当前目录中打开一个名为practice的新文件时，你应该输入：

```
$ vi practice
```

因为这是个新文件，缓冲区是空的且屏幕的显示如下：

```
~
~
~
"practice" [New file].
```

最左边的波浪符（~）表示文件中没有文本，连空白行都没有；底下的提示行（也称为状态行）显示了文件的名称与状态。

你也可以编辑任何目录中的现有文本文件，只要指定文件名即可。假设有一个Unix文件位于/home/john/letter。如果你已经在/home/john目录中，可以使用相对路径名称，例如：

```
$ vi letter
```

即可把letter文件的内容显示在屏幕上。

如果你位于其他的目录下，则需使用完整路径名称：

```
$ vi /home/john/letter
```

打开文件时发生的问题

- *调用vi时，出现*[open mode]*消息。*

  你的终端类型可能未正确识别。立刻输入:q离开这个编辑会话(editing session)。检

查$TERM环境变量，它应该设置为你的终端名称。也可以查询你的系统管理器，以提供正确的终端类型设置。

- *见到下列任何一种消息*：

```
Visual needs addressable cursor or upline capability
Bad termcap entry
Termcap entry too long
terminal: Unknown terminal type
Block device required
Not a typewriter
```

可能是终端类型没有定义，或是terminfo或termcap项中有错误。输入:q离开。检查$TERM环境变量，或要求系统管理为你的环境选择一个终端类型。

- *当你认为文件已存在时，却出现*[new file]*消息*。

确认文件的大小写是否正确（Unix会区分文件名的大小写）。如果正确，很可能位于错误的目录。输入:q离开。检查是否位于正确的目录中（在Unix提示符号下输入pwd）。如果位于正确的目录中，则检查目录中的文件列表（使用ls），以确定是否有个名称只差一点点的文件。

- *调用*vi，*却得到*:*提示符（表示你在ex行编辑模式中）*。

你可能在vi重绘屏幕前将其中断了。在ex提示符（:）下输入vi以进入vi。

- *出现以下消息之一*：

```
[Read only]
File is read only
Permission denied
```

“Read only”表示你只能查看文件，而不能保存任何更动。你可能在查看模式（*view mode*，使用view或vi-R）中调用了vi，或者你对文件没有写入的权限。请参阅第18页的“保存文件时发生的问题”一节。

- *出现以下消息之一*：

```
Bad file number
Block special file
Character special file
Directory
Executable
Non-ascii file
file non-ASCII
```

你要调用来编辑的文件不是常规的文本文件。输入:q! 离开，再检查你要编辑的文件，可以使用 file 命令。

- *当你遇到上述困难而输入*:q*后，却出现如下消息*：

```
No write since last change (:quit! overrides)
```

表示你更改了文件而不自知。输入:q!离开vi。你所做的改变将不会保存到文件中。

运作模式

前面曾经提过，当前“模式（mode）”的概念对vi的运作而言是最基础的。模式有两种：命令模式（command mode）与插入模式（insert mode）。一开始是命令模式，此时所有的按键都代表命令；而在插入模式中，你输入的东西都成为文件的内容。

有时，你可能意外地进入插入模式；或是反过来，意外地离开插入模式。无论何种情况，输入的内容可能影响文件，但不是你想要的结果。

按下ESC键会强制vi进入命令模式。如果你已经处于命令模式，vi会在你按下ESC键时发出“哔”声（因此命令模式有时被称为“哔哔模式”）。

一旦安全地进入了命令模式，即可动手修复意料之外的改变并继续编辑文本。

保存与离开文件

你可以随时离开正在工作的文件，保存编辑结果，并回到Unix提示符号下。vi用于离开并保存编辑结果的命令是ZZ。请注意ZZ这两个字母均为大写。

假定你创建了一个名为practice的文件用于练习vi，并输入了6行文本。想保存这个文件时，首先按下ESC键以检查是否处于命令模式，而后输入ZZ。

按键	结果
ZZ	"practice" [New file]6 lines,320 characters 输入ZZ（写入并保存的命令）后，你的文件会保存成常规的Unix文件
ls	ch01 ch02 practice 列出目录中的文件，显示新创建的practice文件

你也可以用ex命令保存编辑结果。输入:w是保存（写入）文件但不离开vi；若无编辑动作，可输入:q退出；输入:wq，则是保存编辑结果并离开vi（:wq与ZZ相等）。我们会在第五章中完整解释如何使用命令，现在，你只需要记住一些写入与保存的命令即可。

结束而不保存编辑结果

在初学vi时，如果你很喜欢大胆地做各种尝试，有两个ex命令可以帮你轻松地将文件恢复到原来的样子。

当你想要消除所有的编辑结果，回到原来的文件时，可采用命令：

:e! ENTER

将恢复上一次存储的文件内容，这样你可以从头再来。

假设你想消除所有的编辑结果，然后直接离开vi，可采用命令：

:q! ENTER

将离开你所编辑的文件并回到Unix提示符下。有了这两个命令后，所有上一次存档后仍在缓冲区的编辑动作都会消失。vi一般不会让你放弃编辑结果。在:e与:q命令后的感叹号可使vi不理会这个禁令，即使缓冲区有所改变，仍然会执行这两个命令。

保存文件时发生的问题

- *尝试写入文件，却出现以下的消息：*

  ```
  File exists
  File file exists - use w!
  [Existing file]
  File is read only
  ```

  输入:w! *file*以覆盖现有的文件，或者输入:w *newfile*将编辑的结果写入新的文件。

- *你想写入文件，却没有写入的权限，并得到"Permission denied"消息。*

  使用:w *newfile*将缓冲区写入一个新文件。如果你拥有目录的写入权限，则可使用mv将原来的文件用新文件替换。如果你没有目录的写入权限，则输入:w *pathname/file*，将缓冲区写入某个你拥有写入权限的目录（如你的主目录或是/tmp）。

- *尝试写入文件，却得到文件系统已满的消息。*

  输入:!rm *junkfile*来删除一些不需要的（大）文件，空出一些空间（在感叹号后开始ex命令便可以使用 Unix）。

  或者输入:!df看看其他文件系统有没有空间。如果有的话，则在其他文件系统中选择一个目录，用:w *pathname*写入你的文件（df是检查磁盘剩余空间的Unix命令）。

- *系统进入开放模式并且显示文件系统已满。*

  vi用于临时文件的磁盘已满。输入:!ls /tmp查看有无可以移除的文件，以腾出一些空间（注6）。如果有的话，先创建一个临时的Unix shell，以便移除文件或

注6：vi可能把临时文件放在/usr/tmp、/var/tmp或你的当前目录中，你可能需要查找是哪里的空间不够了。Vim的临时文件则通常与被编辑的文件处于同一个目录。

发出其他Unix命令。你可以输入:sh，创建一个shell；输入CTRL-D或exit，结束shell并回到vi（在现在的Unix系统上，当使用作业控制的shell时，你只需要输入CTRL-Z来暂停vi，即可回到Unix提示符，再输入fg即回到vi）。腾出空间之后，使用:w!写入文件。

- *尝试写入文件，却得到磁盘限额已满的消息。*

  试试:pre（:preserve的简写）命令强迫系统保存你的缓冲区。如果没有用，看看有没有文件可以移除。使用:sh（如果使用的是作业控制系统，也可以用CTRL-Z）暂时离开vi并移除文件。完成之后，使用CTRL-D（或fg）回到vi，再用:w!写入文件。

练习题

学习vi唯一的方法就是练习。你已经知道如何创建新文件以及回到Unix提示符。创建一个名为practice的文件，插入一些文字，接着保存并结束此文件。

在当前目录中打开一个名为 practice 的文件：	vi practice
插入文字：	随便输入一些文字
回到命令模式：	ESC
结束vi，保存编辑结果：	ZZ

第二章

简单的文本编辑

本章介绍如何用vi来编辑文本，我们将编写成教案。你会在本章中学到如何移动光标并做简单的编辑。如果你从来没有用过vi，请把整章读完。

后续各章将扩展你的视野，让你能进行速度更快、功能更强的编辑。vi对老手来说，最大的优点之一是有许多的选项可用；对新手来说，最大的缺点之一则是有太多不同的编辑命令。

你不应该死背所有的vi命令，应先从本章中介绍的基本命令开始学起，并注意各命令间的共通模式。

在学习vi时，你应该尽量寻找那些可以交给编辑器的工作，然后找出对应的命令。虽然在后面的章节你会学到高级的vi功能，但是在学习跑之前，应该先学会走路。

本章内容包括：

- 移动光标
- 增加与更改文本
- 删除、移动与复制文本
- 更多进入插入模式的方法

vi命令

vi有两种模式：命令模式与插入模式。你刚进入vi时，是处于命令模式，编辑器正在等你输入命令。命令可以让你移动到文件的任何地方进行编辑，或者进入插入模式以增加新文本。命令也可以用来结束编辑（保存或忽略编辑结果），回到Unix提示符。

你可以将两种模式想象成两套不同的键盘。在插入模式中，键盘就像是打字机；而在命令模式中，每一个键都有新的意义或用于开始某些指令。

有好几种方法可以用于告诉vi你要进入插入模式，最常见的一种就是按下i。i不会显示在屏幕上，但是按下它以后，所有你输入的东西都会显示在屏幕上并且会进入缓冲区。光标会置于当前插入的点（注1）。如果要告诉vi你想停止插入文本，则按下ESC，光标将往回移动一个字符（因此会位于你输入的最后一个字符上）并回到vi的命令模式。

例如，你打开了一个新文件并想插入“introduction”这个词。依序输入iintroduc-tion后，屏幕上的显示将是：

```
introduction
```

当你打开新文件时，vi会进入命令模式，并将第一个按键（i）当作插入命令；其后所有的按键都被视为文本，直到按下ESC为止。如果需要在插入模式中修改错误，只要按退格键后将错误去掉即可。依照终端类型的不同，退格键可能会消除原来输入的文字，也可能只是后退而已。无论如何，退格键经过的内容都将被删除。请注意，你不能用退格键到达进入插入模式的那一点之前。

vi有一个选项可以让你设置右边界，并于到达右边界时自动插入换行符（carriage return）。但现在，当你插入文本时，先按ENTER以断行。

有时你会不知道是位于插入模式还是命令模式。当vi所做的事不合乎你的预期时，先按ESC一两次，查看你位于何种模式。如果听到“哗”声，表示位于命令模式。

移动光标

你在做编辑工作时，可能只有一小段时间是在插入模式下增加新文本，大部分的时间都是在编辑现有的文本。

在命令模式中，你可以将光标移到文件的任何地方。因为在做简单的编辑（更改、删除与复制文本）时，光标需先移到要更动的文本上，你会想要将光标尽快地移到那个地方。

vi移动光标的命令包括：

- 上、下、左、右键——一次一个字符（character）。
- 前进或后退一个文本块（text block）——一次一个单词、句子或段落。

注1：有些版本会于状态行显示正处于插入模式。

- 在文件中一次一屏（screen）地前进后退。

图2-1中，下划线指示了光标现在的位置，圆圈表示各种vi命令会将光标移动到的地方。

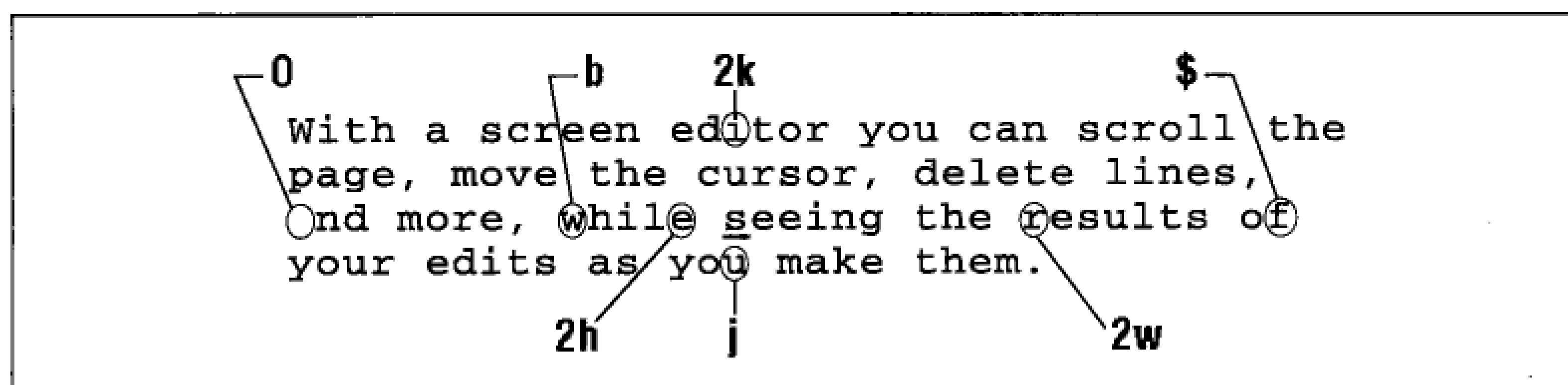


图2-1：简单的移动命令

单一的移动

h、j、k、l，这4个位于键盘中心的键可以移动光标：

h

向左一个字符

j

向下一行

k

向上一行

l

向右一个字符

你也可以用方向键（←、↓、↑、→）、+与-键做上下移动，或用到ENTER与BACKSPACE键，但是这种方法比较少用。一开始，你也许会觉得用字母代替方向键来移动光标很奇怪，然而过一会，你便会发现这是vi最令人喜爱的一点——不需要让你的手指离开键盘的中心，就可以到处移动光标。

在移动光标前，先单击ESC，以确定处于命令模式。使用h、j、k、l前后移动光标。当你往一个方向移动到极限时，会听到蜂鸣声，而光标会停止移动。例如，当你位于一行的开头或是结尾时，就不能用h或l移动到上一行或是下一行，必须使用j或k（注2）。同样地，你不能将光标移到超过波浪符（~）的地方，就是没有文本的行，也不能移到第一行文本之前。

注2： Vim若设置有nocompatible，则可使用l或空格键“越过”一行的结尾而进入下一行。

数值参数

你可以在移动命令的前面加上数字。如图2-2所示，命令4l把光标向右移4个字符，就像按了4次l一样（llll）。

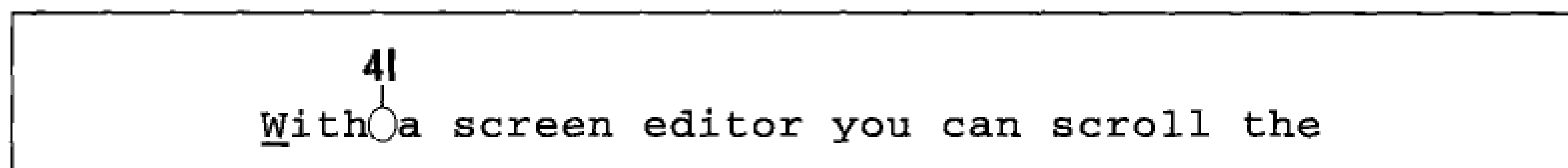


图2-2：通过数字多次键入命令

这种可将命令重复多次的功能会给你更多的选择，并增强命令的功能。接下来，在接触更多命令时，也应该牢记这一点。

在一行中移动

保存了practice文件后，vi会显示一个消息，告诉你这个文件中有多少行。这里的行不一定与屏幕上可见的行（通常限制在80个字符）一样长，而是表示在换行符（newline）间的任何文本（换行符会在你处于插入模式并按下ENTER键时被插入文件）。如果你输入200个字符后才按下ENTER键，vi就把所有200个字符当成同一行（即使这200个字符在屏幕上看起来是好几行）。

前面提过，vi可以设置与右边界的距离，以便自动插入换行符。这个选项是 wrapmargin（简写为wm），你可以将它设为10个字符：

```
:set wm=10
```

这个命令不会影响已经输入的行。在第七章中将提到如何设置选项。（但这个选项真的不能等到那时才介绍！）

如果你没有设置vi的wrapmargin（环绕）选项，则必须手动使用换行符来换行，让一行的长度保持在可以接受的范围。

有两个用于在一行中移动的命令是：

0

移到一行的开头

$

移到一行的结尾

下面的范例中，显示了行编号（使用number选项可在vi中显示行号，在命令模式中输入:set nu即可设置这个选项。详细内容将于第七章中介绍）。

```
1    With a screen editor you can scroll the page,
2    move the cursor, delete lines, insert characters,
     and more, while seeing the results of your edits
     as you make them.
3    Screen editors are very popular.
```

其中逻辑上的行数（3）和屏幕上见到的行数（6）是不一样的。如果光标位于*delete*的*d*上时，你输入了$，则光标会移到*them*后面的句点上；如果输入0，光标会回到第二行开头*move*的*m*上面。

按照文本块来移动

你可以依照文本块来移动光标：单词、句子或是段落都可以。w命令使光标移动一个单词，符号与标点也算是一个单词。以下呈现w的光标移动：

```
cursor, delete lines, insert characters,
```

你也可以不算符号与标点，只将光标移动到下一个单词，此时要用W命令（可以想象成“大的”或“大写的”Word）。

使用 W 做光标移动的结果如下：

```
cursor, delete lines, insert characters,
```

若想后退一个单词，应该使用b命令。大写的B则是倒退一个单词，但不把标点、符号视为单词。

前面提过，移动命令可接受数值参数，因此不管是w或b命令，都可以用数值将其重复多次。2w把光标往前移动2个单词；5B会往回退5个单词，但不算标点、符号。

若想移动到特定一行时，你可以使用G命令。只按G将移动到文件的结尾，1G则可到达文件的顶端，42G可移动到第42行。在第50页的“G（转至）命令”一节中将有更详细的说明。

我们将在第三章中讨论如何依照句子与段落移动光标。现在，先练习你所知道的光标移动命令，并且加上数值参数。

简单的编辑

当你在文件中输入文字时，很少一次就能尽善尽美，可能会有错字或者想修订词句；有时你的程序也会有错误。在输入文本之后，你应该可以对它们更改、删除、移动或复制。图2-3显示了我们可能想做的文件编辑工作，并且用校对符号标示出来。

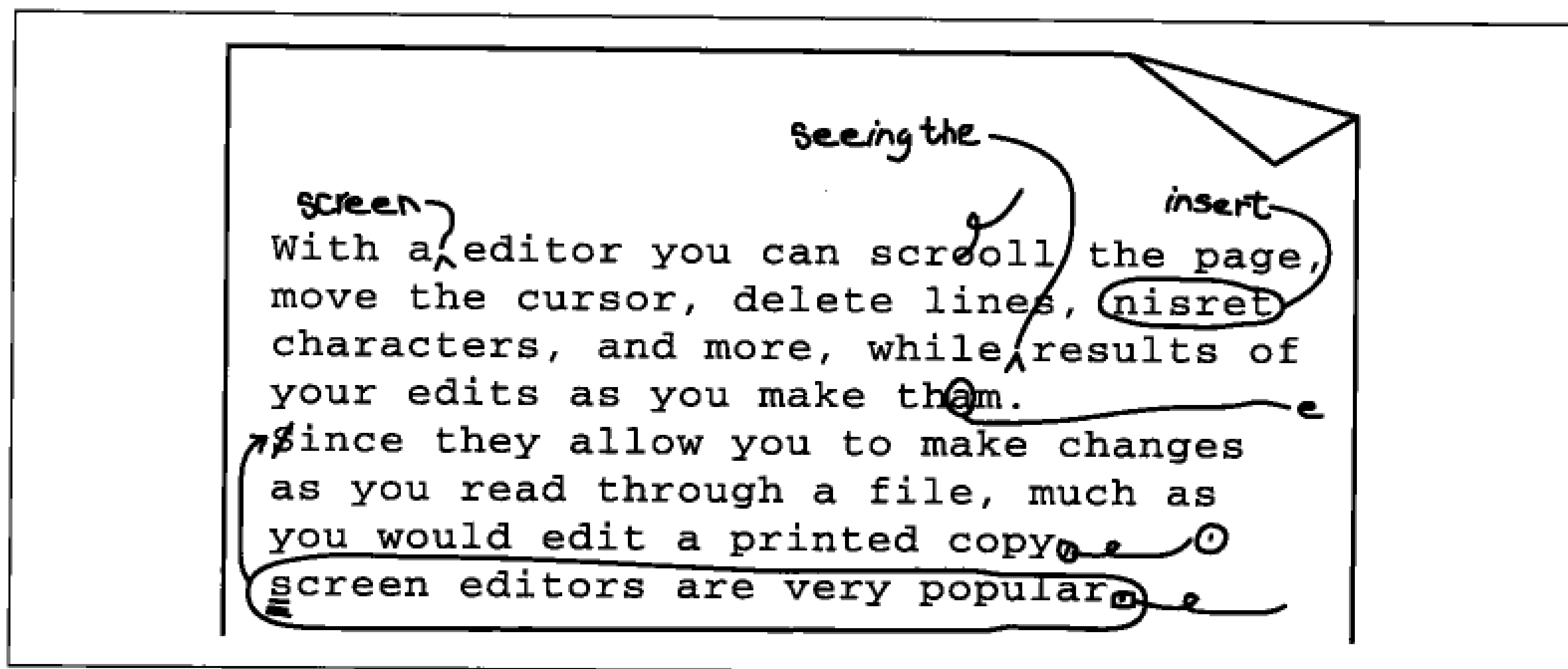


图2-3：校对编辑

在vi中，只要几个简单的键，就可以做出这些动作：i表示插入，a表示添加，c表示更改，d表示删除。移动与复制文本，则需要用到两个命令。移动时先用d做删除，再用p进行放置；复制时先用y做“拖曳”的动作，再用p进行放置．这些编辑动作将在后续章节中一一解说。图2-4把图2-3中标示的校对动作显示为相应的vi命令。

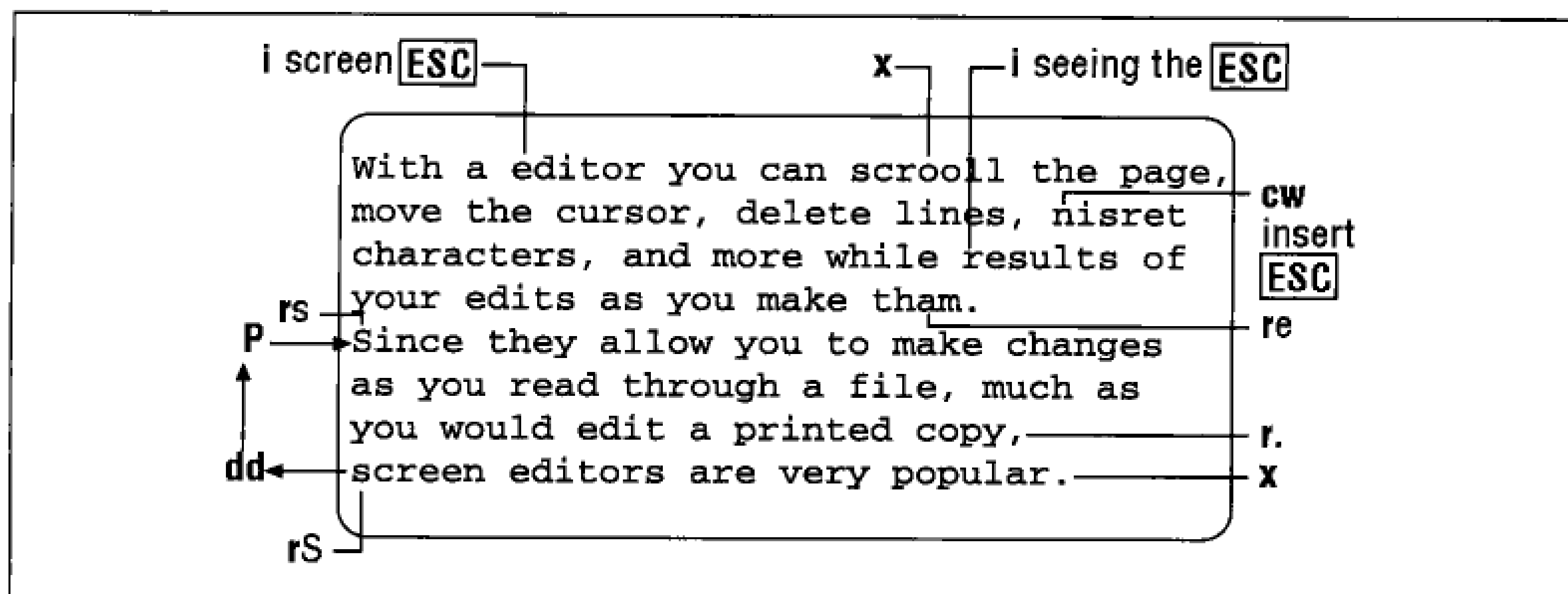


图2-4：以vi命令进行编辑

插入新文本

你已经见过用来在新文件中输入文本的插入命令。但你也会在编辑现有的文本时使用插入命令，以加入漏掉的字符、单词或句子。假设practice文件有下列句子：

```
you can scroll
the page, move the cursor, delete
lines, and insert characters.
```

光标的位置如上图所示。要在这个句子的开头插入"With a screen editor"，则需要如下的输入：

按键顺序	结果

2k	`you can scroll`
	`the page, move the cursor, delete`
	`lines, and insert characters.`
	用k命令将光标往上移两行，移到想做插入动作的文本行：

iWith a	`With a you can scrooll`
	`the page, move the cursor, delete`
	`lines, and insert characters.`
	单击i进入插入模式，并开始输入文字。

screen editor ESC	`With a screen editor you can scroll`
	`the page, move the cursor, delete`
	`lines, and insert characters.`
	输入完毕后，按ESC结束输入动作并回到命令模式。

附加文本

你可以在文件中任何地方用附加命令a来附加文本。a与i的用法几乎都相同，除了前者是在光标后面插入文本，而后者是在光标之前插入文本。你可能注意到，当按下i进入插入模式后，光标不会移动，除非插入一些文本。然而，当你按下a进入插入模式时，光标会往右移一个字符，你输入的文本将在原来的光标位置的后面出现。

更改文本

需替换文件中的文字时，可以用更改命令c。为了告诉c有多少文本需要更改，可以把c与光标移动命令一起使用。此时，将把光标移动命令视为命令c会影响的文本对象。例如，c可以用来更改光标所在位置的文本：

cw
: 从光标到这个单词的结尾。

c2b
: 从光标往前两个单词。

c$
: 从光标到本行结尾。

c0
: 从光标到此行的开头。

在下达更改命令后，你可以将标示出的文本用任何数量的新文本替换，像全部空白、一个单词或是几百行文本都可以。c与i、a一样，在按下ESC键前，都会停留在插入模式。

当这些更改只影响当前这一行时，vi将在被更改文本的结尾显示$，因此你可以看到这一行的哪些部分将被影响（参阅以下的cw范例）。

单词

要更改一个单词，可用c（更改）命令结合表示单词（word）的w命令。你可以将一个单词换成（cw）更长或更短的单词（或任何长度的文本）。cw可被想成“删除标示出的单词，再插入新的单词，直到按下ESC为止”。

假设文件practice中有如下文本行：

```
With an editor you can scroll the page,
```

如果想将*an*改为*a screen*，你只需要更改一个单词：

按键顺序	结果
w	`With an editor you can scroll the page,` 用w将光标移动到你想开始编辑的地方。
cw	`With a$ editor you can scroll the page,` 下达更改单词的命令。被更改的文本结尾会以$标示出。
a screen	`With a screen editor you can scroll the page,` 输入更改的文本，再按ESC回到命令模式。

cw也可以用于更改单词的一部分。例如，要将*spelling*改成*spelled*，可以将光标移到i上，输入cw，再输入ed，接着用ESC结束。

整行

如果要更改当前这一整行的内容，有一个特殊的更改命令：cc。cc会将一整行换成任何输入的文本，直到按下ESC为止。原本的光标位置并不重要，cc会直接换掉整行文本。

cw这一类命令的工作方式与cc不同。使用cw时，原来的文本会先留着，直到输入内容逐渐将它覆盖掉，而任何余下的原文本（到$为止的文本），在你按下ESC后立即消失。但使用cc时，原文本会先消失，留下一行空白用来插入文本。

vi命令的一般形式

就当前提到的更改命令而言，你可能发现了如下模式：

(*command*)(*text object*)

*command*部分是更改命令c，*text object*（文本对象）则是光标移动命令（输入时不需加上括号）。但c不是唯一需要文本对象的命令。d（删除）命令、y（拖曳）命令都适用这种形式。

另外要记住的是，光标移动命令可使用数值参数，因此可将数值加在c、d、y等命令的文本对象上。例如d2w与2dw都是删除两个单词的命令。记住这一点后，你会发现大部分的vi命令遵循如下模式：

(*command*)(*number*)(*text object*)

或者相等的模式：

(*number*)(*command*)(*text object*)

它们的工作方式是这样的：*number*与*command*为可选项。如果没有这两部分，只是单纯的光标移动命令；如果加上*number*，则出现移动多次的效果；结合*comman*（c、d、y等等）与*text object*，则会得到编辑命令。

当你认识到这些组合的多样性后，vi就成为有强大威力的编辑器了！

前者用在任何影响范围不到一整行文本的更改，而后者则用在影响一行文本以上的更改。

C可更改光标所在位置到此行结尾间的文本。它的功用和c加上特殊行末指示符$的效果一样（c$）。

cc与C命令实际上是其他命令的简写，因此不符合vi命令的一般形式。在讨论删除与拖曳命令时，会提到更多的简写。

字符

另外一种更改的方法是r命令。r把一个字符替换成另一个，结束后，不需要按ESC回到命令模式。假设下面这一行文本有一个拼错的地方：

```
With a screen editor you can scroll the page,
```

只有一个字符需要更改。在这里你不想使用cw，因为你不想重打整个单词。可用r命令来更改光标所在处的字符：

按键顺序	结果
rW	`With a screen editor you can scroll the page,` 使用r命令加上要更改的字符W。

替换文本

假设你只想更改几个字符，而不是一整个单词，使用替换命令（s）即可实现。其本身会替换一个字符，而在前面加上数值后，即可替换许多个字符。像更改命令（c）一样，被替换的最后一个字符会用$标示出，以便查看有多少文本会被替换。

S命令，就像其他的大写命令，可以更改一整行文本。它与C命令略有不同——C命令会更改从光标所在位置直到该行结尾间的文本，S命令则不管光标位于哪里，都会整行删除。然后vi把光标移到此行开头并进入插入模式。在S命令前面加上数值可以替换许多行。

s与S都会让你进入插入模式，而当你结束新文本的输入时，按ESC。

R命令和对应的小写命令一样，用来替换文本。不同之处在于，按下R会进入覆盖模式，你输入的字符将逐一覆盖屏幕上的字符，直到按下ESC为止。R命令最多只能覆盖一整行，当按下Enter时，vi会打开新行并进入插入模式。

更改大小写

更改字母的大小写是一种特殊的替换。波浪号（~）命令可将小写字母改成大写的，或者将大写字母改成小写的。将光标移到你要更改的字母上并输入~，这个字母的大小写就会改变，而光标移到下一个字符。

在旧版的vi中，你不能为~指定数值参数或文本对象，而新的版本已经接受数值参数了。

如果你要一次改变一行以上的文本的大小写，则必须使用Unix命令（如tr）来过滤文本，这将在第七章中介绍。

删除文本

文件中文本的删除可使用d命令。就像更改命令，删除命令也需要文本对象（用于表示要处理的文本量）。你可以删除单词（dw）、删除一行（dd与D），或使用稍后学到的其他移动命令。

删除文本时，先将光标移到删除开始的地方，再输入删除命令（d）及文本对象，例如用w表示单词。

单词

假设某文件中有下列文本：

```
Screen editors are are very popular,
since they allowed you to make
changes as you read through a file.
```

光标的位置如上所示，你想要删除第一行中的一个*are*。

按键顺序	结果

2w	`Screen editors are are very popular,`
	`since they allowed you to make`
	`changes as you read through a file.`
	光标移到要开始删除的地方（*are*）。

dw	`Screen editors are very popular,`
	`since they allowed you to make`
	`changes as you read through a file.`
	下达删除单词的命令（dw）删除*are*。

dw会删除光标所在位置的单词，请注意单词后的空白也会被删除。

dw也可以用来删除单词的一部分。在下例中：

```
since they allowed you to make
```

你想删除单词*allowed*后面的*ed*。

按键顺序	结果
dw	`since they allowyou to make` 输入删除单词的命令（dw），将从光标所在位置开始的部分单词删除。

dw总是会将此行中下一个单词前的空白删除，但是我们在上一个例子中并不希望如此。若要留下单词间的空白，应该使用de，它只会删除到单词本身的结尾；dE删除的范围则是到包括标点在内的单词结尾。

你也可以向前删除（db）或者删除到一行的结尾或开头（d$或d0）。

整行

dd命令删除光标所在行的一整行内容，它并不会只删除部分行。像cc一样，dd是个形式特殊的命令。利用前面范例中的文本，并将光标放在第一行，如下所示：

```
Screen editors are very popular,
since they allow you to make
changes as you read through a file.
```

你可以删除开头两行：

按键顺序	结果
2dd	`changes as you read through a file.`
	输入删除两行的命令（2dd）。请注意，光标虽然并非位于一行的开头，仍会删除一整行。

D命令会删除从光标所在位置到本行结束间的文本（D是d$的简写）。例如，光标位于如下所示的位置：

```
Screen editors are very popular,
since they allow you to make
changes as you read through a file.
```

你可以从光标开始删除文本到本行结束：

按键顺序	结果

D	`Screen editors are very popular,`
	`since they allow you to make`
	`changes`

	下达删除光标右侧所有部分的命令（D）。

字符

通常我们只想删除一个或两个字符。就像有替换单一字符的特殊更改命令r，当然也有特殊的删除命令x，它只会删除光标所在位置的字符。在下面这一行中：

```
You canzmove text by deleting text and then
```

你可以按下x以删除z（注3）。大写的X则删除光标前的字符。在这些命令前面加上数值可以删除多个字符。例如，5x会删除从光标所在之处开始往右的5个字符。

删除文件时发生的问题

- *你误删了文本，想要补救。*

  有好几种方法可以恢复被删除的文本。如果你刚刚删除一些东西，但马上就发觉了，只要输入u就可以撤销上一个命令（如dd）。但这只适用于尚未下达其他命令的时候，因为u只会撤销最近一个命令。另一方面，U会恢复一整行成原来面貌，就是做任何改变之前的样子。

  你也可以使用p命令，恢复最近几次的删除动作，因为vi会将最近9次的删除动作保存在9个编号的删除缓冲区中。举个例子，如果你知道要恢复的缓冲区是第三个，则可以输入：

  ```
  "3p
  ```

  把第三个缓冲区“放到”光标所在的下一行上。

  但这只对删除的一整行才有用。被删除的单词或是一行的一部分都不会被保存到缓冲区中。如果你要恢复一个单词或一整行的一部分，而且u命令没有用，请单独使用p命令。它会恢复所有你刚刚删除的内容。后续章节会谈到更多关于命令u与p的内容。

  请注意，Vim支持“无限”恢复（infinite undo），挽救失误因此简单多了。请参考第297页的“撤销‘撤销’”一节，了解更多信息。

移动文本

在vi中，采用“删除后再置放文本”的方法使文本移动，就像使用“剪切与粘贴”一样。每次移动一个文本块时，文本块会先被删除，再存储在特殊的缓冲区中；接着移动到另一个位置，并被放置命令（p）放在新的位置。虽然移动对一整行文本比单词有用，但还是可以移动任何的文本块。

放置命令（p）会将缓冲区的文本放在光标后，大写的P命令则把文本放置在光标前。如果你移动了一行以上的文本，p命令把移动的文本放在光标后的新一行（或很多行）；如果你移动的文本不到一行，p命令把移动的文本放在光标之后的同一行。

注3： 为什么用字母x呢？可能是用打字机来“消除”（x-ing out）错误的意思吧，但现在谁还在用打字机呢？

假设practice文件中有下列文本：

```
You can move text by deleting it and then,
like a "cut and paste,"
placing the deleted text elsewhere in the file.
each time you delete a text block.
```

你想将第二行的*like a "cut and paste,"*放到第三行之后，使用删除就可以做到：

按键顺序	结果

dd	`You can move text by deleting it and then,`
	`placing the deleted text elsewhere in the file.`
	`each time you delete a text block.`
	在光标位于第二行时，会删除这一行。这些文本会被放到缓冲区中（保留的内存）。

p	`You can move text by deleting it and then,`
	`placing that deleted text elsewhere in the file.`
	`like a "cut and paste"`
	`each time you delete a text block.`
	使用p（放置）命令，在光标所在行的下一行恢复被删除的行。不过要完成这个句子的重组，可能还需要更改大小写与标点符号（使用r命令），以对应新的句子结构。

注意： 一旦删除文本后，你必须在发出下一个更改或删除命令前恢复文本。如果你做了另一个会影响缓冲区的编辑动作，删除的文本就会消失。你可以一直重复放置的动作，只要不做新的编辑动作即可。在第四章中，你会学到如何将文本存入被命名的缓冲区中，以便稍后访问。

对调两个字母

你可以用xp（删除一个字符，再放到光标后面）来对调两个字母。例如，在*mvoe*这个词中，*vo*的位置应该相反才对。要更正这个错误，将光标放在*v*上并按下x，接着按p。可用*transpose*（对调）这个单词帮助记忆xp这个命令：x表示*trans*，而p表示*pose*。

现在我们暂不讨论对调两个单词的命令。在第七章第111页中的“更多映射按键的范例”一节讨论了一些命令的集合，可以用来对调两个单词。

复制文本

将一段文本复制下来再于别的地方使用，常常可以节省编辑（与按键）的时间。使用两个命令——y（拖曳）与p（放置），就可以复制任何数量的文本，并放置到另一个地

方。拖曳命令会将选中的文本放到特殊的缓冲区中，一直保留到下一个拖曳命令（或删除命令）发生为止。你可以用放置命令将这些文本放到文件的任何地方。

就像更改与删除命令一样，拖曳命令可与任何光标移动命令合并使用（如yw、y$、4yy）。拖曳命令最常用在一行（或更多行）文本上，因为拖曳并放置一个单词通常比直接插入单词还要花时间。

快捷键yy用于拖曳一整行的文本，就像dd与cc一样。但是，另一个快捷键Y因为某些原因，它的操作方法并不像D与C，不会拖曳从光标所在位置到一行结尾的文本，而是拖曳一整行，即Y与yy的行为完全一样。

假设practice文件中有以下文本：

```
With a screen editor you can
scroll the page.
move the cursor.
delete lines.
```

你想要造出三个完整的句子，都以*With a screen editor you can*作为开头。此时不需要在文件中到处移动或重复地输入相同文段，而可以使用拖曳与放置命令。

按键顺序	结果

yy	`With a screen editor you can`
	`scroll the page.`
	`move the cursor.`
	`delete lines.`
	将你要复制的文本拖曳到缓冲区。光标可以位于你想拖曳的行中任意一处（或许多行中的第一行）。

2j	`With a screen editor you can`
	`scroll the page.`
	`move the cursor.`
	`delete lines.`
	将光标移到要放置被拖曳文本的地方。

P	`With a screen editor you can`
	`scroll the page.`
	`With a screen editor you can`
	`move the cursor.`
	`delete lines.`
	用P命令将被拖曳的文本放到光标所在行的上一行。

按键顺序	结果（续）
jp	With a screen editor you can scroll the page. With a screen editor you can move the cursor. With a screen editor you can delete lines. 将光标往下移一行，再将被拖曳的文本用p命令放置到光标所在行的下一行。

拖曳与删除命令共享同一个缓冲区。每一个新的删除或拖曳动作都会覆盖缓冲区的内容。在第四章中将看到，放置命令最多可用9个拖曳或删除缓冲区中的内容。你也可以直接将拖曳与删除的内容放到26个已命名的缓冲区，以便处理多个文本块。

重复或撤销上一个命令

每一个编辑命令均存储到一个临时的缓冲区，直到发出下一个命令为止。例如在某个单词后插入*the*。用于插入的命令与插入的文本都会暂时被存储起来。

重复

想要重复相同的编辑命令时，可以使用重复命令——句号（.），以节省时间。将光标移到欲重复前一编辑命令的地方，再输入一个句号。

假设你的文件有如下内容：

```
With a screen editor you can
scroll the page.
With a screen editor you can
move the cursor.
```

在删除一行后，你只要输入句号即可再删除另一行。

按键顺序	结果
dd	With a screen editor you can scroll the page. move the cursor. 用dd删除一行。
.	With a screen editor you can scroll the page. 重复删除。

旧版本的vi在重复命令上有问题。例如，在设置了wrapmargin的情况下，这些版本可能在重复长文本的插入时发生困难。如果你使用的是这种版本的vi，迟早会发现这个问题的害处。此时你也没有办法补救，不过事先得到警告总是好的（新版似乎没有这个问题）。有两个方法可以预防在长文本插入时可能发生的问题。你可以在重复插入命令之前先写入文件（:w），以便在出问题时取回原来的文件；或把wrapmargin设置取消，如下所示：

```
:set wm=0
```

在第七章的“更多映射按键的范例”一节，我们会展示一个简易解决wrapmargin问题的方案。在某些版本的vi中， 会重复最近一个插入命令。于插入模式下输入CTRL-@，结束时将进入命令模式。

撤销

前面提过，如果出了错误操作，可以撤销上一个命令。只要按下u即可，光标不需要在原来下命令时所在的位置。

我们继续上一个例子，恢复在practice文件中所删除的行：

按键顺序	结果
u	With a screen editor you can scroll the page. move the cursor.

u会撤销上一个命令并恢复删除的行。

大写字母U会撤销所有对同一行的编辑动作，只要光标还在这一行即可。一旦你移到别的行，就不能使用U了。

注意，你可以用u撤销上次“撤销”的动作。u也可以撤销U；而U会撤销所有在同一行中的更改，包括u在内。

注意： 小秘诀：因为u可撤销自己，这样就产生一个在文件中移动的巧妙方式。如果你想回到上一次暂停编辑的地方（称为A点），只要撤销即可。当你再撤销这次撤销动作时，仍会待在A点，而不会跳到撤销文本的最末端。

Vim让我们用CTRL-R“重做”（redo）一次撤销操作。结合无限次撤销，你将可在文件更改的历史记录中倒退或前进。请参考第297页的“撤销‘撤销’”一节以获得更多信息。

更多插入文本的方法

你可以用以下命令在光标之前输入文本：

    itext to be inserted ESC

你也可以用a命令在光标后插入文本。还有其他插入命令可以在光标的不同位置插入文本：

A
: 在一行的结尾处附加文本

I
: 在一行的开头处插入文本

o
: 在光标所在位置的下一行打开新行

O
: 在光标所在位置的上一行打开新行

s
: 删除光标所在位置的字符后再替换文本

S
: 删除一整行后再替换文本

R
: 用新的字符覆盖现有的字符

上述所有命令都会让你进入插入模式。在插入文本后，记得按ESC回到命令模式。

A（附加）与i（插入）可以节省进入插入模式前将光标移到一行开头或结尾所花的时间（A命令比$a省下一次按键。虽然一个按键可能不算什么，但是在你越来越熟悉编辑工作与不耐烦时，你会想省略更多按键）。

o与O（打开）可以节省按下回车键的动作。你可以在一行的任何位置使用这些命令。

s与S（代换）可以让你删除一个字符或一整行，并用任何数量的文本代换。s等效于c加上两个SPACE键，而S与cc等效。s最方便的用处之一是将一个字符换成多个字符。

R在你想更改文本但不知道确实数量时很有用。例如，你不用猜是用3cw或4cw，只要输入R后再输入要替换的文本即可。

插入命令的数值参数

除了o与O，以上的插入命令（包括i与a）都接受数值参数。通过数值前缀你可以用I、I、a、A等命令插入一整行的下划线或其他字符。例如，输入50i*ESC会插入50个星号；输入25a*-ESC则会插入50个字符（25对星号与连字符的组合），这样就不必重复输入许多字符了（注4）。

使用数值参数后，r会将许多字符替换成重复的单一字符。例如，在C或C++代码中，如果要将||换成&&，只要将光标放在第一个|字符上，再输入2r&。

你也可以在S前加入数值参数以替换许多行。不过，使用c加上光标移动命令更为快速，并且更有灵活性。

有个适合使用数值参数加上s命令的例子：当你要更改单词中的几个字符时，输入r是不适合的，输入cw又会变更太多文本，而数值参数加上s的作用通常与R一样。

当然还有其他的命令可以自然地组合起来。例如，ea可以用来将新的文本加到一个单词的后面。训练自己记得这些常用的命令组合，变成反射动作，对你将大有帮助。

用J合并两行

在编辑文件时，有时在最后会剩下一些短行，造成阅读困难。当你需要将两行合并成一行时，可将光标移到第一行的任何地方，然后按J来合并两行。

假设你的practice文件内容如下：

```
With a
screen editor
you can
scroll the page, move the cursor
```

按键顺序	结果

J	`With a screen editor`
	`you can`
	`scroll the page, move the cursor`
	J将光标所在的行与下一行合并。

.	`With a screen editor you can`
	`scroll the page, move the cursor`
	重复上一个命令（J），继续把下一行与当前这一行合并。

注4： 版本非常老的vi在插入超过一行文本时会出现问题。

在J前使用数值参数可以将多个连续的行合并起来。在上一个例子中，使用命令3J可合并三行。

问题集

- *你输入命令时，文字出乎意料地在屏幕上到处乱跑，一切的运作都不如预期。*

  确定你输入了j，而不是输入了J。

  你可能按了CAPS LOCK键却没有注意到。vi对大小写很敏感，也就是说，大写命令（I、A、J）与小写命令（i、a、j）是不一样的，因此按下CAPS LOCK后输入的所有命令都会被当成大写命令。请按CAPS LOCK回到小写状态，再按ESC确定你处于命令模式，然后可输入U以恢复上一行的改变，或者输入u以撤销上一个命令。你可能需要做一些额外的工作，才能恢复刚才弄乱的部分。

基本 vi 命令的复习

表2-1呈现了一些结合c、d、y与各种文本对象的命令。最后两行显示了其他编辑命令。表2-2与表2-3列出了其他基本命令。表2-4总结了其他本章提到的命令。

表2-1：编辑命令

文本对象	更改	删除	复制
一个单词	cw	dw	yw
两个单词，不包括标点符号	2cW或c2W	2dW或d2W	2yW或y2W
后退三个单词	3cb或c3b	3db 或 d3b	3yb或y3b
一整行	cc	dd	yy或Y
到一行的结尾	c$或C	d$或D	y$
到一行的开头	c0	d0	y0
单个字符	r	x或X	yl或yh
五个字符	5s	5x	5yl

表2-2：光标移动命令

移动	命令
←、↓、↑、→	h、j、k、l
到下一行的第一个字符	+
到上一行的第一个字符	-
到单词的结尾	e或E

表2-2：光标移动命令（续）

移动	命令
往前一个单词	w或W
往后一个单词	b或B
到一行的结尾	$
到一行的开头	0

表2-3：其他操作

操作	命令
往缓冲区中放置文本	P或p
打开vi，如果指定了文件则打开文件	vi *file*
保存编辑结果，并离开文件	ZZ
不保存编辑结果，并离开文件	:q!

表2-4：文本创建与操纵命令

编辑动作	命令
在光标所在位置插入文本	i
在一行的开头插入文本	I
在光标所在位置附加文本	a
在一行的最后附加文本	A
在光标下一行打开新行	o
在光标上一行打开新行	O
删除一行并替换文本	S
用新文本覆盖现有的文本	R
合并当前这一行与下一行	J
切换当前字符的大小写	~
重复上一个动作	.
撤销上一个动作	u
将一整行恢复到原来的状态	U

会使用上述列表中的命令，就可以算是学会vi了。但是，要释放vi的真正威力（并增加你的生产力），你需要更多的工具，接下来几章会介绍这些工具。

第三章

快速移动位置

你不会只用vi来创建新文件，而是会花很多时间来编辑现有的文件。你也很少会想从文件的第一行开始逐行地移动，而是会想要到文件的一个特定位置，然后开始工作。

所有的编辑动作都是将光标移动到你想开始编辑的地方（如果用ex行编辑器命令，则需确定要编辑的行号）。本章展示了如何以各种方式（根据屏幕、文本、模式或行号）移动。在vi中移动的方法有许多种，而这些方法的编辑速度取决于移动到目的地时所需的按键数。

本章内容包括：

- 根据屏幕来移动
- 根据文本块来移动
- 根据搜索模式的结果来移动
- 根据行号来移动

根据屏幕来移动

当你在看书时，你会以页码确定在书中的“位置”，例如，你停下来的那一页或是索引中的某个页码。在编辑文件时，可没有那么方便：有些文件只有几行，一眼就可以看完；但是许多文件却有几百（甚至几千）行！

你可以将文件想象成有文字的长卷轴，而屏幕是一个窗子，（通常）可以显示其中24行的文字。

在插入模式中，如果输入了一整屏的文字并输入到屏幕最底行的结尾，这时按下ENTER，则最上面一行就看不见了，同时会有一行空白行出现在屏幕的最下面，用来输入新的文字。这称为滚动（scrolling）。

在命令模式中，你可以往前或往后滚动，以便观察文件中的任何文字。而且，光标移动命令可加上数值参数，以便快速地在文件各处移动。

滚动一整屏

以下是在文件中往前或往后滚动整屏或半屏的vi命令：

^F
: 往前滚动一整屏

^B
: 往后滚动一整屏

^D
: 往前（下）滚动半屏

^U
: 往后（上）移动半屏

（其中的^符号表示CTRL键。所以，^F表示按着CTRL键并同时按下f键。）

还有两个命令分别可将屏幕往上滚动一行（^E）或往下滚动一行（^Y）。然而，这两个命令不会将光标移到一行的开头。在执行命令后，光标会出现在上（下）一行中的同一个地方。

用z重新调整屏幕位置

如果要往上或往下滚动屏幕，但是又想让光标维持在原来的文本行，就可以用z命令：

z *ENTER*
: 将光标移到屏幕顶端并滚动屏幕

z.
: 将光标移到屏幕中心并滚动屏幕

z-
: 将光标移到屏幕底端并滚动屏幕

使用z命令加上数值参数可重复多次移动，但其实没有意义（毕竟你只需要将光标移到屏幕的顶端一次即可。重复相同的z命令并不会再做任何移动）。然而z可以接受行号作为数值参数，指定行将成为新的当前位置。例如，zENTER是将当前行移到屏幕顶端，但是200zENTER则会把第200行移到屏幕顶端。

重画屏幕

有时当你正在编辑时，计算机会显示一些消息在屏幕上．这些消息不是编辑缓冲区的一部分，但却会影响你的工作。当系统消息出现在屏幕上时，你需要重画屏幕。

由于当你滚动屏幕时，就会重画一部分（或全部）的屏幕，因此你只需要上下滚动一次，就可以消除那些多余的消息。当然你也可以输入CTRL-L，只重画而不滚动。

在屏幕中移动

你可以保持当前屏幕，或保留文件当前的视图，而在屏幕范围中移动光标：

H

移到屏幕顶端的行。

M

移到屏幕中央的行。

L

移到屏幕底端的行。

*n*H

移到屏幕顶端往下的第*n*行。

*n*L

移到屏幕底端往上的第*n*行。

H可以将光标从屏幕的任何地方移到第一行，也就是该页第一个字符的位置（home）；M会将光标移到中间那一行；而L会移到最后一行。要将光标移到第一行下面一行，则使用2H。

按键顺序	结果
L	With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of your edits as you make them. Screen editors are very popular, since they allow you to make changes as you read through a file. 用L命令将光标移到屏幕的最后一行。

按键顺序	结果（续）
2H	With a screen editor you can Scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of your edits as you make them. Screen editors are very popular, since they allow you to make changes as you read through a file. 用2H命令将光标移到屏幕的第二行。

根据行移动

在当前屏幕中，也有以行为标准进行移动的命令。你已经知道j与k命令了，还可以用如下命令：

ENTER

  移到下一行的第一个字符。

+

  移到下一行的第一个字符。

-

  移到上一行的第一个字符。

上面三个命令会将光标往下或往上移一行，并置于该行第一个字符上，空格或tab（制表符）会被忽略。而j与k会将光标移动到上一行或下一行的第一个位置（假设光标原本位于第一个位置），即使那个位置是空格也一样。

在当前行移动

不要忘了h与l可将光标往左与往右移动，而0与$会将光标移到一行的开头与结尾。你也可以用如下命令：

^

  移到当前行的第一个非空格处。

n|

  移到当前行的第 *n* 列。

就像上面的行移动命令一样，^会移到当前行的第一个字符处，忽略任何空格与制表符。相对地，0则会把光标移到当前行的第一个位置处，即使是空格也一样。

根据文本块来移动

另一个在vi文件中移动的方式，则是以文本块为单位——单词、句子、段落或小节。

你已经知道如何以一个单词为单位往前或往后移动（w、W、b、B）。另外，你也可以用下面的命令实现光标移动：

e
: 移到单词的结尾。

E
: 移到单词的结尾（忽略标点符号）。

(
: 移到当前句子的开头。

)
: 移到下一个句子的开头。

{
: 移到当前这一段的开头。

}
: 移到下一段的开头。

[[
: 移到当前这一节的开头。

]]
: 移到下一节的开头。

vi会寻找?、.、!这些标点符号，以辨认句子的结束。当这些标点符号后面有至少两个空格或是作为一行的最后一个非空格字符时，vi则将其定位为一个句子的结束。如果你在句号后面只留了一个空格或该句以引号结束，则vi不能辨认这个句子。

“段落”的定义是下一个空白行前的文本，或是出现在troff MS macro package中默认的段落宏（.IP、.PP、.LP、.QP）前的任何文本。同样地，“小节”的定义是下一个默认的节宏（.NH、.SH、.H 1、.HU）前的文本。:set命令可用于自定义这些被当成段或节分隔符的宏，这在第七章中另有介绍。

请记住，你可以把移动命令加上数值参数。例如，3)会往前移动三个句子。另外你也可以用移动命令编辑文件：d)会删除文本直到当前这个句子的结尾，而2y}会向前复制（拖曳）两段文本。

根据搜索模式的结果来移动

在大文件中移动时，最快速的方法之一乃是搜索一串文本，更正确地说，是搜索特定模式（pattern）的字符。有时搜索操作可以用来找寻拼错的单词或是代码中某个变量出现的所有地方。

搜索命令是特殊字符 /（斜线）。当你输入斜线时，其会出现在屏幕底端，接着再输入要搜索的特定模式，格式为：*/pattern*。

模式可以是一个完整的单词，或是一连串字符（称为"字符串"）。例如搜索单词*red*，搜索结果会找出整个单词*red*，但是也会找出occur*red*。如果你在模式前或模式后加上一个空格，则这个空格会被当成单词的一部分。模式输入完毕后，按ENTER结束命令（与所有出现在底端的命令一样）。vi像所有其他的Unix编辑器一样，也有一套特别的模式匹配语言，可以让你寻找变动的文本模式：例如，任何以大写字母开头的单词，或是作为一行开头的单词*The*。

我们会在第六章提到更强大的模式匹配语法。而现在，只要把模式当成一个单词或词组就可以了。

vi会从光标所在位置开始往前搜索，如果需要则绕回文件的开头。光标会移到要搜索的模式第一次出现的地方。如果没有找到，状态行会出现"Pattern not found"的消息（注1）。

同样，利用practice文件，示范如何依据搜索结果来移动光标：

按键顺序	结果
/edits	With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of your edits as you make them.
	搜索模式*edits*。按ENTER以输入此模式，然后光标会直接移到这个模式上。
/scr	With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of your edits as you make them.
	搜索模式*scr*，按ENTER以输入此模式。请注意搜索的模式*scr*后没有空格。

注1：实际的消息依照不同的vi同类品而有所不同，但意义是一样的。一般而言，我们不会每次都不厌其烦地提醒"消息文字可能有所不同"，因为实际上的意义都是相同的。

搜索会绕回文件的开头。还有，你可以搜索任意组合的字符串，不一定要是完整的单词。

要往回搜索，则不是输入/，而是输入?：

> ?*pattern*

在这两种搜索命令中，如果需要的话，都会绕回文件的开头或结尾。

重复搜索

上一次所搜索的模式会留在你的编辑会话中。搜索过后，若要再次搜索上一个模式，不用重复原来的按键顺序，你可以用一些命令来重复搜索：

n

　　往同一个方向重复搜索。

N

　　往相反的方向重复搜索。

/ENTER

　　往前重复搜索。

?ENTER

　　往后重复搜索。

因为上一个模式仍然可用，你可以搜索一个模式后做些其他工作，接着再用n、N、/或?来搜索同一个模式，而不用重新输入。搜索的方向会显示在屏幕底端（/是往前，? 是往后）。（nvi不会显示n与N命令的搜索方向。Vim会把要搜索的文本也放在命令行里，让我们能用上下键滚动查看搜索命令的历史记录。）

继续上一个范例。因为scr这个模式仍然可以用来搜索，你可以执行如下操作：

按键顺序	结果
n	With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of your edits as you make them.
	用n（下一个）命令移到下一个模式*scr*出现的地方。
?you	With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of your edits as you make them.

按键顺序	结果（续）
	用?从光标位置开始往前搜索到第一个出现*you*的地方。你需要在输入模式之后按ENTER。
N	With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of your edits as you make them. 重复上一个搜索命令，但是方向相反（往前）。

有时你只想搜索在光标位置之前的单词，并不想让搜索绕回文件的开头。vi有一个选项wrapscan，可以控制搜索是否要绕回开头。你可以取消这个功能：

```
:set nowrapscan
```

若设置了nowrapscan，而往前的搜索失败时，状态行会显示如下消息：

```
Address search hit BOTTOM without matching pattern
```

若设置了nowrapscan，而往后的搜索失败时，状态行显示的消息将以“TOP”取代“BOTTOM”。

通过搜索修改文本

/与?搜索运算符可与更改文本的命令结合，例如与c和d结合。延续前例做过的改变：

按键顺序	结果
d?move	With a screen editor you can scroll the page, your edits as you make them. 从光标位置开始向前删除到出现*move*的地方。

请注意，此处的删除是以字符为基础，而不是删除整行。

这一节只介绍了搜索模式最基本的命令。第六章会提到更多关于模式匹配的内容以及模式匹配在对文件做整体改变时的用途。

在当前行中搜索

搜索命令也可用于一行内的搜索。fx命令会将光标移到下一个出现*x*字符的地方（其中*x*代表任何字符）。tx命令则将光标移到下一个出现的*x*字符的前一个字符，可以用分号继续搜索出现的*x*字符。

下面列出了行内搜索命令，这些命令都不会把光标移到下一行。

fx
: 搜索（将光标移到）本行中下一个出现*x*的地方，*x*代表任何字符。

Fx
: 搜索（将光标移到）本行中上一个出现*x*的地方。

tx
: 搜索（将光标移到）本行中下一个出现*x*的地方的前一个字符。

Tx
: 搜索（将光标移到）本行中上一个出现*x*的地方的后一个字符。

;
: 重复上一个搜索命令，方向相同。

,
: 重复上一个搜索命令，方向相反。

在这些命令前加上数值*n*，则会搜索*x*字符第*n*次出现的地方。假设你在编辑practice文件的这一行：

```
With a screen editor you can scroll the
```

按键顺序	结果
fo	`With a screen editor you can scroll the` 搜索本行中第一个出现的*o*。
;	`With a screen editor you can scroll the` 用 ; 命令移到下一个出现*o*的地方（搜索下一个*o*）。

dfx会删除到下一个*x*字符为止的所有文本，包括*x*字符在内。这个命令在删除或拖曳一行的一部分时很有用。如果在文本中夹有符号或标点而难以计算单词数量时，你可能需要用dfx代替dw。t命令与f命令很像，不同之处在于它把光标放在要搜索的字符之前。例如，命令ct可用于更改一个句子的内容，而留下最后的句号。

根据行号来移动

文件中的每一行都会依序编号，可借由指定行编号来移动到文件各处。

行号在辨认一大块文本的开始与结束时很有用。行号对程序员也很有用，因为编译器的错误消息会引用到行号。ex命令也会用到行号，我们将在下一章学习。

如果要依照行号来移动位置，必须有指定行号的方式。使用:set nu选项（第七章会再提到），可以在屏幕上显示行号。在vi中，你也可以在屏幕底端显示现在光标位置的行号。

CTRL-G命令会在屏幕底端显示信息，包括当前的行号、文件的总行数以及当前位置占全文的百分比。以practice文件为例，可能显示：

```
"practice" line 3 of 6 --50%--
```

这个信息不管是在显示行号以用于命令，还是在你忘记位置时指示方向都很有用。

依照你所使用的vi版本不同，有可能显示更多的信息，例如光标位于第几个词或文件是否被修改过但还没保存等等，实际的信息格式也会有所不同。

G（转至）命令

你可以用行号在文件中移动光标。G（转至）命令接受行号为数值参数，并直接移到指定行。例如，44G会将光标移到第44行的开头。没有指定行号的G则会移到文件的最后一行。

输入两个反引号（``）会回到原来的位置（即上一次使用G命令时所在的位置），除非你在中间做了编辑操作。如果你做了编辑操作，然后用G以外的命令移动了光标，则``会将光标移回到你上一次做编辑操作时的地方。如果你使用了搜索命令（/或?），``则会带你回到上一次使用搜索命令的地方。一对引号（''）与两个反引号的功用一样，不同之处在于，它把光标移到前次位置所在行的开头，而不是确实的位置上。

CTRL-G显示的总行数可以用来估计要移动多少行。如果你位于有1 000的行文件中的第10 行：

```
"practice" line 10 of 1000 --1%--
```

而你知道要编辑的地方靠近文件的末端，便可以大致估计目的地：800G。

依照行号来移动光标是在一个大文件中快速移动的方法。

vi移动命令的复习

表3-1总结了本章所提到的命令。

表3-1：移动命令

移动	命令
往前滚动一整屏	^F
往后滚动一整屏	^B
往前移动半屏	^D
往后移动半屏	^U
往前滚动一行	^E
往后滚动一行	^Y
将当前行移到屏幕顶端并滚动屏幕	z ENTER
将当前行移到屏幕中心并滚动屏幕	z.
将当前行移到屏幕底端并滚动屏幕	z-
重画屏幕	^L
移到home位置（屏幕的顶端）	H
移到屏幕中间那一行	M
移到屏幕的底端	L
移到下一行的第一个字符	ENTER
移到下一行的第一个字符	+
移到上一行的第一个字符	-
移到当前行的第一个非空格字符	^
移到当前行的第*n*个字符	*n*|
移到单词的结尾	e
移到单词的结尾（忽略标点符号）	E
移到当前句子的开头	(
移到下一个句子的开头	)
移到当前这一段的开头	{
移到下一段的开头	}
移到当前这一节的开头	[[
移到下一节的开头	]]
往前搜索模式	/*pattern*
往后搜索模式	?*pattern*
往同一个方向重复搜索	n
往相反方向重复搜索	N
往前重复搜索	/

表3-1：移动命令（续）

移动	命令
往后重复搜索	?
搜索当前行中下一个出现x的地方	fx
搜索当前行中上一个出现x的地方	Fx
搜索当前行中下一个出现x的地方的前一个字符	tx
搜索当前行中上一个出现x的地方的后一个字符	Tx
重复上一个搜索命令，方向相同	;
重复上一个搜索命令，方向相反	,
转至第n行	nG
转至文件结尾	G
回到上一个记号或上下文	``
回到包含上一个记号的行的开头处	''
显示当前的行号（不是移动命令）	^G

第四章

越过基础的藩篱

我们已经介绍了基本的vi编辑命令，如i、a、c、d与y。本章则对这些你已经知道的编辑命令加以扩展，包括：

- 描述额外的编辑工具，并以一般的命令形式作复习。
- 其他进入vi的方法。
- 利用缓冲区来存储拖曳与删除的文本。
- 在文件中对你的位置做标记。

更多命令组合

在第二章中，你学到了c、d、y 这些编辑命令以及如何将它们与移动命令及数值相结合（如2cw或4dd）。在第三章中，又增加了许多移动命令。虽然将编辑命令与移动命令结合起来已经不是新鲜事，但我们仍要在表4-1再一次列出这些你已经知道的命令。

表4-1：更多的编辑命令

更改	删除	复制	从光标位置到……
cH	dH	yH	屏幕顶端
cL	dL	yL	屏幕底端
c+	d+	y+	下一行
c5|	d5|	y5|	本行的第5列（第5个字符）
2c)	2d)	2y)	往下第二个句子
c{	d{	y{	上一段
c/*pattern*	d/*pattern*	y/*pattern*	*pattern*（模式）
cn	dn	yn	下一个模式

表4-1：更多的编辑命令（续）

更改	删除	复制	从光标位置到 ……
cG	dG	yG	文件结尾
c13G	d13G	y13G	第13行

请注意表4-1列出的组合都符合一般形式：

（*number*）（*command*）（*text object*）

其中*number*是可有可无的数值参数，而*command*是c、d或y，*text object*则是一个移动命令。

vi命令的一般形式在第二章中讨论过，你可以复习一下表2-1与表2-2。

打开vi的选项

到当前为止，你都是用以下的命令打开vi：

```
$ vi file
```

还有其他很好用的vi命令打开选项。你可以从某一行或以某个模式来打开一个文件，你也可以用只读方式打开一个文件。还有选项能在系统死机后恢复你正在编辑的文件中所有的更改。

前进到特定的位置

当你开始编辑现有文件时，可以先读入文件，再移动到某个模式（pattern）第一次出现的位置或移动到某一行。你也可以在命令行指定第一次移动的方式，例如依搜索结果或行号移动（注1）。

$ vi +*n* *file*
在第*n*行打开 *file*。

$ vi + *file*
在最后一行打开*file*。

$ vi +/*pattern* *file*
在第一个出现*pattern*的地方打开*file*。

注1：根据POSIX标准，vi应该用-c *command*，而不是这里显示的+c *command*。但一般来说，为了向下兼容性，两种写法都能接受。

在文件practice中，如果要打开文件并直接跳到包含单词*screen*的地方，只要输入以下命令：

按键顺序	结果
vi +/screen practice	With a screen editor you can scroll the page, move the cursor, delete lines, and insert characters, while seeing the results of your edits as you make them. Screen editors are very popular, since they allow you to make changes as you read
	即将vi命令加上选项+/*pattern*（此例为*screen*），就可直接跳到包含*screen*的文本行。

如上例所示，搜索的模式不必位于屏幕的顶端。如果你的模式中包含空格，则需用单引号或双引号括起整个模式（注2）：

```
+/"you make"
```

或者用反斜线将空格转义：

```
+/you\ make
```

另外，如果你要使用第六章中描述的一般模式匹配语法，可能必须用单引号或反斜线将一个或多个特殊字符保护起来，以防止shell将其解释成其他意思。

+/*pattern*在你编辑到一半却必须离开时很有用。你可以将当前位置用某个模式标记起来，如ZZZ或HERE。当你回来时，只要记得/ZZZ或/HERE就可以了。

注意： 当你在vi中编辑时，一般都已打开wrapscan选项。如果你把环境自定义为wrapscan禁用时（请参考第47页的“重复搜索”小节），可能不能使用+/*pattern*。这时如果你用这种方法打开文件，vi会将光标置于文件最后一行，并显示“Address search hit BOTTOM without matching pattern.”（已经寻找至文件末尾，找不到匹配的模式。）。

只读模式

有时候你想查看文件，但是又不想在无意间更动文件（例如读入一个很长的文件，用于练习vi的移动命令，或只是想在脚本文件中上下滚动）。此时可以用只读模式打开文件，这时仍可使用所有的vi移动命令，但是不能更改文件内容。

注2： 这是shell的限制，不是vi的限制。

要用只读模式显示文件，应输入以下命令：

```
$ vi -R file
```

或是:

```
$ view file
```

（view命令也像vi命令，可以使用任何跳到文件中特定地方的命令行选项（注3）。）如果你决定要改变文件内容，可以在write命令（:w）后面加上感叹号，以覆盖掉只读模式：

```
:w!
```

或是：

```
:wq!
```

如果你在写入文件时发生问题，可以参考附录C中的问题集。

恢复缓冲区

在你编辑文件时，系统有时可能发生错误，通常，在你上一次存档后所做的更动都会消失。然而，有一个选项-r可以恢复系统死机时的编辑缓冲区。

如果使用传统Unix系统与原始的vi，会于系统重新启动后第一次登录时收到一个邮件消息，表示缓冲区已经保存起来。而且，如果你输入命令：

```
$ ex -r
```

或是：

```
$ vi -r
```

将得到系统保存下来的所有文件列表。

用-r选项加上文件名可以恢复编辑缓冲区。例如，要恢复系统死机时practice文件的编辑缓冲区，可输入：

```
$ vi -r practice
```

此时最好立刻恢复原来的文件，以免你在无意中又对文件做了编辑，结果你必须在保存的缓冲区内容与新做编辑的文件间解决新旧版本的问题。

注3： 通常view只是对vi的链接。

使用:pre命令，你可以强制系统即使没有死机也保存缓冲区。如果你编辑了某个文件，后来发现因为没有写入权限而不能保存时，这就很有用了（你可以将这些内容写入另一个文件或是另一个你可以写入的目录中。参阅第一章中的“保存文件时发生的问题”一节）。

注意： 各种vi同类品的恢复动作不尽相同，甚至同一产品的不同版本间也可能有差别，最好要查阅特定版本的帮助文件。vile不支持任何形式的恢复。vile的说明文档中推荐我们使用autowrite与autosave选项。第七章的“自定义 vi”一节中会提到如何实际操作。

善加利用缓冲区

你已经知道，编辑时的最后一次删除（d或x）或拖曳（y）的内容会保存到缓冲区中（内存中的一块）。你可以访问这些缓冲区并使用放置命令（p或P）将这些保存的文本放回文件中。

vi会将最后9次的删除操作保存在编号的缓冲区中，你可以访问其中任何一个，以恢复任何一次（或所有）的删除操作（然而小规模的删除，如一行中的一部分，并不会保存到编号的缓冲区中。这些删除只能在刚做操作后立刻用p或P命令恢复）。

vi也可以让你将拖曳（复制）的文本放在依字母标识的缓冲区中。拖曳的文本可以被保存到26个缓冲区（a～z）中，并且可在编辑会话的任何时候，使用放置命令来恢复这些文本。

恢复删除

一次删除大块文本很方便也很好用，但如果不小心删除了53行重要的数据该怎么办呢？有一个办法可以恢复你的前9次删除，因为它们都保存在编号的缓冲区中。最后一次删除的内容存在缓冲区1，倒数第二次的则存在缓冲区2……

要恢复删除操作，先输入"（双引号），接着指定缓冲区编号，再使用放置命令。以恢复倒数第二次的删除（位于缓冲区 2）为例：

```
"2p
```

缓冲区2包含的删除的内容将出现在光标之后。

如果不确定哪一个缓冲区包含了要恢复的文本，你也不用一直重复输入"*n*p。如果在恢复（p）一次后利用重复命令（.）做恢复，缓冲区的编号便会自动增加，再加上用u撤销恢复，即可用下例搜索编号的缓冲区：

"1pu.u.u *依此类推*

上例可逐一把每个缓冲区的内容放置到文件中。每一次输入u时，恢复的文本会被移除。输入点号（.）时，则把下一个缓冲区的内容恢复到文件中。不断输入u与 .，直到找到所需的文本为止。

将文本拖曳到命名缓冲区中

稍早我们学习到，在做任何编辑前，必须先放置（p或P）未命名缓冲区的内容，不然缓冲区的内容就会被覆盖掉。你可以用y与d搭配26个专门给复制与移动文本使用的命名缓冲区（a～z）。如果你使用命令缓冲区存放被拖曳的文本，便可以在任何时间取回其中的内容。

要将文本拖曳到命名缓冲区，需在拖曳命令前加上双引号（"）以及缓冲区名称（以字符表示）。例如：

"dyy　　*将当前行拖曳到缓冲区d中*
"a7yy　　*将后续7行拖曳到缓冲区a中*

拖曳的内容放至命名缓冲区，移动光标到新的位置后，使用p或P可将文本取回：

"dP　　*将缓冲区d的内容放置在光标前*
"ap　　*将缓冲区a的内容放置在光标后*

要将缓冲区文本的一部分取出来，在当前是做不到的，只能选择全部取回或完全不取回。

在下一章中，你会学到如何编辑多个文件。当你知道如何不离开vi也可在文件间切换时，便可以使用命名缓冲区在文件间传送部分文本了。使用其他vi同类品的多窗口编辑功能时，非命名缓冲区也可用于在文件间传送数据。

你也可以将删除的文本保存到命名缓冲区中，方法是一样的：

"a5dd　　*将删除的5行保存到缓冲区a中*

如果你用大写字母指定缓冲区名称，则拖曳或删除的文本会被附加到当前缓冲区中。因此可以选择性地做移动与复制。例如：

"zd)

删除从光标处开始到所在句子的结尾处的内容，并将内容保存到缓冲区z中。

2)

往前移两个句子。

"Zy)

将下一个句子添加到缓冲区z中。你可以继续在某个命名缓冲区中添加更多的文本，但请注意：如果你一时忘记，在拖曳或删除文本到缓冲区时没有用大写字母指定缓冲区名称，则会把缓冲区的内容覆盖掉，前面已有的文本都会消失。

对一处做标记

在一个vi会话中，你可以在某处做一个看不见的“书签”，然后在别处编辑，完成后再回到“书签”位置。可在命令模式中输入：

m*x*

将当前位置标记成 *x*（*x*可以是任何字符）。

'*x*

（单引号）将光标移到标记*x*所在行的第一个字符。

`*x*

（反引号）将光标移到以*x*标记的字符。

``

（两个反引号）在移动位置之后，回到上一个标记或上下文的确切位置。

''

（两个单引号）回到上一个标记或上下文所在行的开头。

注意：标记只有在当前vi会话中有用，并不会存储在文件中。

其他高级编辑技巧

你可以用vi执行其他高级编辑操作，但是要先学一些ex编辑器的用法，这在下一章会提到。

vi缓冲区与标记命令的复习

表4-2总结了所有版本的vi通用的命令行选项。表4-3与表4-4总结了缓冲区与标记命令。

表4-2：命令行选项

选项	意义
+*n file*	在第*n*行打开*file*
+ *file*	在最后一行打开*file*

表4-2：命令列选项（续）

选项	意义
+/*pattern file*	在第一个出现*pattern*的地方打开*file*
-c *command file*	在打开文件后执行命令；通常是行号或搜索模式（POSIX 版本的是 +）
-R	用只读模式打开（与view命令相同）
-r	在死机后恢复文件

表4-3：缓冲区名称

缓冲区名称	缓冲区用途
1～9	最后9次删除操作的内容，从最后一次到最先一次
a～z	你可以使用的命名缓冲区，大写字母表示附加到缓冲区现有内容后

表4-4：缓冲区与标记命令

命令	意义
"*b command*	用缓冲区*b*执行命令
m*x*	将当前位置标记为*x*
'*x*	将光标移到标记*x*所在行的第一个字符
`*x*	将光标移到用*x*标记的字符
``	回到上一个标记或上下文的确切位置
''	回到上一个标记或上下文所在行的开头



第五章

ex编辑器概述

这是一本vi的工具书，为什么我们要用一整章来介绍另一个编辑器ex呢？其实ex并不算是另一个编辑器。vi其实是更通用、更底层的ex行编辑器的“可视模式”，有些ex命令在vi中很有用，可以节省许多编辑的时间。这些命令甚至大部分可在不离开vi的情况下使用（注1）。

你已经知道如何将文件想象成一连串有编号的行。ex可以给你机动性更高、能力更强的编辑命令。利用ex，你可以轻松在文件间移动，并用各种方法传送文本。你可以快速地编辑超过一整个屏幕的文本块。使用全局替换，你可以对整个文件中的某个模式做替换操作。

本章会介绍ex及其命令，你可以学到：

- 使用行号在文件中移动。
- 使用ex命令对一块文本做复制、移动与删除。
- 保存文件与文件的一部分。
- 编辑多个文件（读入内容或命令，在文件间切换）

ex 命令

在vi或任何满屏编辑器发明前，人与计算机需依靠打印终端来沟通，而不是在阴极射线管（CRT）上沟通。（或者在加上定位设备与终端仿真程序的位映射屏幕上沟通）行号可以用于快速地识别工作文件的一部分，而行编辑器是用来编辑这些文件。编程者或其他用户通常会在打印终端上打出一行（或多行），下达编辑命令更动这一行后，再重新打出这一行以供检查。

注1： vile与大多数vi同类品不同，许多高级ex命令不能使用，这些细节会在第十八章中提到。

我们早已不再用打印终端来编辑文件了，但是有些ex行编辑器的命令仍然可以在复杂的可视化编辑器上使用，因为它们是创建在ex之上。虽然用vi来做大部分的编辑都比较简单，但是ex以行为导向，在对文件做大规模的改变时就成了优点。

注意： 在本章中所用到的命令，大部分都以文件名为参数。虽然在文件名中加入空格字符是合法的，但是我们非常不推荐这种行为：ex会严重混淆，而你在尝试让ex接受这种文件名时，会碰上很多麻烦。在分隔文件名中的成分时，最好使用下划线、横线或句号，这样会比较方便使用。

在你开始背ex命令（或者根本就跳过）之前，我们先揭开一些行编辑器的神秘面纱，看看ex如何工作，这样可以稍微了解很难懂的命令语法。

先打开一个你熟悉的文件，用于尝试ex命令。就像你可以用vi编辑器打开文件一样，你也可以用ex行编辑器来打开文件。打开ex时，会见到一些消息，它列出了文件的总行数以及命令提示符（:）。例如：

```
$ ex practice
"practice" 6 lines, 320 characters
:
```

你不会看到文件中的任何一行，除非下达显示一行或多行的ex命令。

ex命令中包含了行地址（可以只是一个行号）以及命令，以换行符结束（按ENTER即可）。一个最基本的命令是p，代表打印（到屏幕上）。因此，假设你在提示符下输入1p，会见到文件的第一行：

```
:1p
With a screen editor you can
:
```

事实上，你可以省略p，因为行号本身与显示出此行的命令是相同的。要显示出多行文本，可以指定一个范围的行号（例如，1,3——两个数值之间用逗号分隔，中间有没有空格都无妨）。例如：

```
:1,3
With a screen editor you can
scroll the page, move the cursor,
delete lines, insert characters, and more,
```

没有行号的命令会被当作只对当前这一行起作用。以替换命令（s）为例，其将一个单词替换成另一个，可以这样输入：

```
:1
With a screen editor you can
```

```
:s/screen/line/
With a line editor you can
```

请注意，更动的行会在命令执行后重新显示出。你也可以换个方法做相同的事：

```
:1s/screen/line/
With a line editor you can
```

即使你是从vi中调用ex命令，而不是直接使用ex，花一点时间了解ex也是很值得的。你会感觉到为什么需要告诉编辑器该操作哪一行（或多行）以及该执行哪一个命令。

在你对practice文件试过一些ex命令后，应该用vi打开同一个文件，这样就可以在比较熟悉的可视模式中查看文件。:vi命令可以让你从ex进入vi。

想在vi中使用ex命令，你必须输入特定字符:（冒号）。输入命令后，按ENTER来执行。因为在ex编辑器中你只要在冒号提示符后输入行号，就可以跳到那一行。所以，想在vi中移到某文件的第6行，请输入：

```
:6
```

然后按下ENTER。

做完下面的练习后，我们就只讨论在vi中执行的ex命令。

练习题：ex 编辑器

在 Unix 提示符下，用ex编辑器打开practice文件：	ex practice
出现消息：	"practice" 6 lines, 320 characters
跳到第一行并打印（到屏幕上）	:1
（在屏幕上）打印第一到第三行：	:1,3
将第一行的*screen*换成*line*：	:1s/screen/line
打开vi编辑器：	:vi
跳到第一行：	:1

问题集

- 在vi中编辑文件时，有时会意外进入ex编辑器。

  在vi的命令模式中输入Q时会调用ex。若是意外进入ex编辑器时，输入命令vi即可回到vi编辑器。

用ex编辑

许多负责常见编辑操作的ex命令在vi中都有更简单的相应命令。删除一个单词或一行时，当然使用dw或dd，而不用ex的delete命令。然而，当你要更改许多行时，会发现ex命令更有用。你可以用一个命令更改一大块文本。

常用的ex命令与其缩写整理如下。但请记住，在vi中输入ex命令时，命令前必须加上冒号。你可以用完整的命令名称或缩写，以好记为原则。

Full name	Abbreviation	Meaning
delete	d	删除行
move	m	移动行
copy	co	复制行
	t	复制行（与co同义）

如果你觉得用空格来分隔ex命令的多个部分会比较容易读，确实可以这么做。例如在行地址（line address）、模式与命令间使用空格分隔。然而，你不能在模式中使用空格区隔，也不能以空格作为替换命令的结尾。

行地址

每一个ex编辑命令都需要知道要编辑的行号。而对ex的move与copy命令来说，还必须提供文本移动或复制的目的地。

指定行地址的方法有下面几种：

- 指定明确的行号
- 用符号来指定相对于当前位置的行号
- 标识某些行的搜索模式作为地址

我们来看一些例子。

定义行范围

你可以用行号来清楚地定义某一行或某段行的范围。明确指定行号的地址称为绝对行地址。例如：

:3,18d

删除第3行到第18行。

:160,224m23

将第160行到第224行移到第23行之后（类似vi中的delete与put）。

:23,29co100

将第23行到第29行复制到第100行之后（类似vi中的yank与put）。

为了简化使用行号的编辑操作，你可以将所有的行号显示在屏幕左边。可使用下面的命令：

```
:set number
```

或简写为：

```
:set nu
```

这时会显示行号，文件practice将显示如下：

```
1  With a screen editor
2  you can scroll the page,
3  move the cursor, delete lines,
4  insert characters and more
```

当你写入文件时，行号并不会写入文件，打印时也不会打印出来。行号只有在结束vi会话或者禁用set选项时才会消失，禁用set选项的命令如下：

```
:set nonumber
```

或是

```
:set nonu
```

要暂时显示某些行的编号时，可以使用#符号，例如：

```
:1,10#
```

即可显示第1行到第10行的行号。

如第三章所述，你可以使用CTRL-G命令显示当前行的编号。因此，欲确认一段文本开头与结尾的行号时，可以通过将光标移动到文本块的开头，键入CTRL-G，再将光标移动到块结尾并键入CTRL-G而得知。

另一个分辨行号的方法是ex的=命令：

:=

列出文件的总行数。

:.=

列出当前所在行的行号。

:/*pattern*/=

列出*pattern*第一次出现时的行号。

行寻址符号

你也可以用符号表示行地址。点号（.）表示当前这一行；$表示文件的最后一行。%表示文件中的每一行，与1,$组合的意义相同。这些符号可与绝对行地址合并使用。例如：

:.,$d

删除当前这一行到文件结尾间的文本。

:20,.m$

将第 20 行到当前这一行间的文本移到文件结尾。

:%d

删除文件中所有的行。

:%t$

将所有的行复制到文件结尾（做连续的复制）。

除了绝对行地址之外，你还可以指定相对于当前这一行的地址。+与-的操作类似于算数操作。放在数值前面时，其表示加上或减去后面的数值。例如：

:.,.+20d

删除当前这一行到20行之后的行之间的文本。

:226,$m.-2

将第226行到文件结尾间的行移到当前这一行的两行之前。

:.,+20#

显示当前这一行及向下20行之间的行号。

事实上，在使用+或-时并不需要输入点号（.），因为当前行会被假定为开始的位置。

如果后面没有接着数值，+与-分别等于+1与 -1（注2）。同样地，++与--分别可将范围增加一行，以此类推。+与-也可以用在搜索模式中，这在下一节会提到。

注2：在相对地址中，你不可将 +或-与它们后面的数值分开。例如，+10表示“后面10行”；但+10则表示“后面11行”（1 + 10行），这可能不是你要的结果。

数字0表示文件的开头（想象中的第0行）。0与1-相同，都可以让你将多行文本移动或复制到文件的开头，也就是第一行文本之前。例如：

`:-,+t0`
: 复制3行（光标上面一行到光标下面一行）并放置到文件的开头。

搜索模式

另一个可以指定行地址的方法是使用搜索模式。例如：

:/*pattern*/d
: 删除下一个包含*pattern*的行。

:/*pattern*/+d
: 删除下一个包含*pattern*的行的下一行（你也可以用+1替代+）。

:/*pattern1*/,/*pattern2*/d
: 从第一个包含*pattern1*的行删除到第一个包含*pattern2*的行。

:.,/*pattern*/m23
: 将当前这一行（.）到第一个包含*pattern*的行之间的文本放到第23行之后。

注意这些模式的前后都要使用斜线作为分界。

如果你在vi与ex中用模式来做删除，两种编辑器的做法不太一样。假设文件practice包含以下几行：

```
With a screen editor you can scroll the
page, move the cursor, delete lines, insert
characters and more, while seeing results
of your edits as you make them.
```

按键顺序	结果
d/while	With a screen editor you can scroll the page, move the cursor, while seeing results of your edits as you make them. vi对模式的删除命令会从光标所在位置删除到*while*这个单词，但是这两行的剩下部分会保留。
:.,/while/d	With a screen editor you can scroll the of your edits as you make them. ex的删除命令会删除指定行的所有文本：在这个例子中包括了当前这一行与包含模式的行，所有的行都会被全部删除。

重新定义当前这一行的位置

有时候在命令中使用相对行位置会产生意料之外的结果。举例来说，假设光标位于第1行，而你想要显示出第100行与它下面的5行，如果输入：

```
:100,+5 p
```

会得到错误消息，表示“第一个地址超过了第二个地址”。这是因为第二个地址是相对于当前光标位置的（第1行），因此命令实际上是表示：

```
:100,6 p
```

这时你需要让命令以为第100行是“当前行”，即使实际上光标是位于第1行。

ex提供了一个方法：当你用分号代替逗号时，第一个行地址会被当成光标当前的地址。例如，这个命令：

```
:100;+5 p
```

即可显示出你所要的结果，此时的+5是相对于第100行计算的。分号对搜索模式的绝对地址很有用。例如，要显示出包含指定模式的下一行及后续10行，可以输入：

```
:/pattern/;+10 p
```

全局搜索

你已经知道如何在vi中使用/（斜线）来搜索文件中的文本模式。ex还有个全局命令g，可以让你搜索模式并显示所有包含这个模式的行。命令:g!的功能则与:g正好相反，它（或是意义相同的:v）用于搜索所有不包含指定模式的行。

你可以将全局命令用在文件中所有的行，或者用行地址将全局搜索限制在一定范围的行中。

:g/*pattern*
　　寻找（移到）文件中最后一次出现*pattern*的地方。

:g/*pattern*/p
　　寻找并显示文件中所有包含*pattern*的行。

:g!/*pattern*/nu
　　寻找并显示文件中所有不包含*pattern*的行，也显示所有找到的行号。

:60,124g/*pattern*/p
　　寻找并显示第60行与第124行之间包含*pattern*的行。

如你所料，g也可以用于全局替换，我们会在第六章中介绍。

合并ex命令

想输入新的ex命令，并不是每次都必须输入冒号。在ex中，竖线（|）可以分隔命令，让你在同一个ex提示符号下合并多个命令（就像在Unix shell提示符下，用分号分隔多个命令一样）。当你使用|时，请记住你所指定的行地址。如果某个命令影响了文件中各行的顺序，下一个命令将在新的行地址上运作。例如：

`:1,3d | s/thier/their/`

删除第1行到第3行（现在位于文件的开头），接着在当前行做替换（即原来的第4行）。

`:1,5 m 10 | g/`*`pattern`*`/nu`

将第1行到第5行移到第10行之后，接着显示所有包含*pattern*的行（包括行号）。

请注意空格的使用，这能让命令更容易理解。

保存与离开文件

你已经学过vi命令中的ZZ，它用于离开vi并写入（保存）文件。但是你常常想用ex命令离开文件，因为这些命令会给你更多的控制权。我们前面曾经提过其中的一些，现在让我们开始正式介绍：

`:w`

将缓冲区中的内容写入（保存）文件中，但不离开。你可以（也应该）在编辑会话里常常使用:w，以保护你的文件免遭系统问题或严重编辑错误的损害。

`:q`

离开编辑器（并回到Unix提示符下）。

`:wq`

写入文件同时离开编辑器。这是无条件写入，即使文件没有被修改也一样。

`:x`

写入文件同时离开编辑器。只有文件被修改过时才会写入（注3）。

vi会保护现有文件以及缓冲区中的编辑工作。例如想把缓冲区中的内容写入现有文件，vi会发出警告。同样地，如果你用vi来打开文件并进行编辑，然后想结束vi，但不保存编辑结果，vi也会产生错误消息：

注3： :wq与:x的差异在编辑源代码并使用make时很重要。make根据文件修改的次数而执行动作。

```
No write since last change.            (自上一次改变之后没有写入。)
```

这些警告消息可以预防许多会造成损失的错误。但有时你会想要强制执行命令，此时在命令后面加上感叹号（!）可忽略警告：

```
:w!
:q!
```

:w!搭配vi -R或view，则可写入以只读模式打开的文件（假设你拥有文件的写入权限）。

:q!是基本的编辑命令，可以离开编辑器而不影响原来的文件。即不管做了怎样的改变，均舍弃缓冲区中的内容。

更改缓冲区名称

你可以用:w将整个缓冲区（你正在编辑的文件副本）以新的文件名保存。

假设你有一个600行的practice文件。你打开文件后做了许多编辑操作，然后想要离开编辑器，但又要同时保存原来的practice文件与编辑后结果，作为比较之用。若将编辑缓冲区中的内容存为名为practice.new的文件，可以用以下命令：

```
:w practice.new
```

则原来的文件practice的内容不会改变（只要你之前没有用过:w）。这时就可以输入:q离开编辑器。

保存一部分的文件

有时你会想将一部分编辑中的文件保存为新文件。例如，你可能已经输入了格式化代码与文本，但想将这些给其他文件使用。

合并使用ex的行寻址命令与写入命令w可以保存一部分文件。以正在编辑practice文件并想将其中的一部分保存成名为*newfile*的文件为例，可以输入：

:230,$w *newfile*
: 将第 230 行到文件结尾保存成名为newfile的文件。

:.,600w *newfile*
: 将光标所在的行到第600行保存成名为newfile的文件。

附加内容到已保存的文件

你可以用Unix的重定向与附加运算符（>>）加上w命令，将缓冲区中的一部分或所有内容附加到现有文件之后。例如输入：

```
:1,10w newfile
```

再输入：

```
:340,$w >>newfile
```

则*newfile*会包含第1—10行以及第340行到缓冲区结尾间的两段内容。

将一个文件复制到另一个文件

有时会想把已经输入完毕的文本与数据复制到当前正在编辑的文件里。在vi中你可以用ex命令来读入另一个文件中的内容：

```
:read filename
```

或简写为：

```
:r filename
```

这个命令把*filename*的内容插入到光标所在位置的下一行。如果你要指定插入位置为其他行，只需在read或r命令前输入行号（或是行地址）即可。

假设你在编辑practice文件并想读入一个位于其他目录下的文件，例如/home/tim中的data文件。则先将光标移到欲插入位置的上一行，再输入：

```
:r /home/tim/data
```

/home/tim/data的内容会被读入到practic中，并从光标所在位置的下一行开始显示出。

要读入同一个文件，但从185行之后插入，则需要输入：

```
:185r /home/tim/data
```

还有其他方法可以读入文件：

$r /home/tim/data
: 将读入的文件放在当前文件的结尾。

:0r /home/tim/data

　　将读入的文件放在当前文件的开头。

:/pattern/r /home/tim/data

　　将读入的文件放在第一个出现*pattern*的行之后。

编辑多个文件

ex命令可以让你在多个文件之间切换，其好处是速度较快。如果你与其他用户共享系统，若每次编辑文件时都要先离开再进入vi，势必很花时间。留在同一个编辑会话并在文件之间切换，不但速度较快，还可以保存你所定义的简写与命令序列（参阅第七章），并且可以保留拖曳缓冲区中的内容，以便在多个文件间复制文本。

用vi同时打开多个文件

当你第一次打开vi时，可以给出多个文件名，接着用ex命令在文件间切换。例如：

```
$ vi file1 file2
```

即可先编辑file1，然后使用ex命令:w写入（存储）file1，此时:n会调用下一个文件（file2）。

假设你要编辑两个文件，practice与note。

按键顺序	结果

vi practice note	`With a screen editor you can scroll`
	`the page, move the cursor, delete lines,`
	`insert characters, and more, while seeing`
	打开practice与note这两个文件。第一个文件practice会出现在屏幕上，可以进行编辑。

:w	`"practice" 6 lines, 328 characters` 用ex的w命令来保存编辑过的practice文件。按ENTER。

:n	`Dear Mr.`
	`Henshaw:`
	`Thank you for the prompt . . .`
	用ex的n命令调用下一个文件note，再按ENTER，开始编辑。

:x	`"note" 23 lines, 1343 characters` 将第二个文件note保存，并离开这个编辑会话。

使用参数列表

ex并不是只能用:n移动到下一个文件。:args参数（简写为:ar）可列出命令行上的文件列表，当前编辑中的文件名称以方括号括起。

按键顺序	结果

vi practice note	`With a screen editor you can scroll`
	`the page, move the cursor, delete lines,`
	`insert characters, and more, while seeing`
	打开practice与note两个文件。其中第一个文件practice会显示在屏幕上。

:args	`[practice] note` vi将参数列表显示在状态行中。以方括号括起当前编辑中的文件的名称。

:rewind（:rew）命令会将当前文件复位成命令行上的第一个文件。elvis与Vim提供了对应的命令:last，用于移动其成命令行上的最后一个。

调用新文件

你不需要在编辑会话刚开始时就调用多个文件，而可以在任何时候用:e命令切换到另一个文件。如果你要在vi中编辑另一个文件，首先必须保存当前文件（:w），再下命令：

:e *filename*

假设你正在编辑文件practice，又想编辑文件letter，则要首先回到practice：

按键顺序	结果
:w	`"practice" 6 lines, 238 characters` 用w来保存practice，再按ENTER。practice被保存，但仍会显示在屏幕上。现在你可以切换到另一个文件了，因为你的编辑结果已经保存。
:e letter	`"letter" 23 lines, 1344 characters` 用e命令调用letter文件，再按下ENTER，开始编辑。

vi会同时“记住”两个文件名，作为当前的与候补的文件名，它们可以用符号%（代表当前的文件名）与#（代表候补的文件名）来表示。#对:e特别有用，因为它可以在两个文件间方便地切换。在上一个例子中，你可以输入命令:e #回到第一个文件practice。你也可以用:r #将practice文件的内容读到当前的文件中。

在当前的文件尚未存储前，vi不会让你用:e或:n来切换文件，除非你特别在命令之后加上感叹号。

例如在编辑过letter之后，你想放弃编辑结果并回到practice文件，可以输入:e! #。

下面的命令也很有用。它会放弃你的编辑结果，并恢复当前文件的上个已保存版本的内容：

```
:e!
```

相对于#符号，%主要用于写出当前缓冲区中的内容到另一个新文件中。例如，前面的“更改缓冲区名称”一节中，我们展示了如何保存practice文件的第二个版本：

```
:w practice.new
```

因为%表示当前的文件名，因此命令也可以写成：

```
:w %.new
```

在vi中切换文件

CTRL ^ 由于切换文件是个常用功能，因此你不需要进入ex命令模式。使用vi的 ^ ^ 命令（控制键Ctrl加上^符号）即可切换文件。这个命令的功能与:e #一样。就像:e命令一样，如果当前缓冲区的内容还没保存，vi不会让你切换到其他的文件。

在文件之间做编辑

当你为拖曳缓冲区命名（名称为一个字母）时，就有了把文本从一个文件移到另外一个的简便方法。使用:e命令将新的文件载入vi的缓冲区时，并不会清除命名缓冲区中的内容。因此，可先在一个文件中拖曳或删除文本（如果需要的话，可存入多个命名缓冲区），再用:e调用新的文件，然后把命名缓冲区的内容放置到新文件中，这样就可以在文件之间传送文本了。

下面的例子说明了如何将文本从一个文件传送到另一个文件。

按键顺序	结果
"f4yy	With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of the edits as you make them 将4行文本拖曳到缓冲区f中。

按键顺序	结果（续）
:w	"practice" 6 lines, 238 characters 保存文件。
:e letter	Dear Mr. Henshaw: I thought that you would be interested to know that: Yours truly, 用:e进入letter文件。将光标移到要放置复制文本的地方。
"fp	Dear Mr. Henshaw: I thought that you would be interested to know that: With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of the edits as you make them Yours truly, 从命名缓冲区f中取出文本并放置到光标处。

另一种将文本从一个文件移到另一个文件的方法是用ex命令:ya（拖动）与:pu（放置）。这些命令分别与vi的y、p命令的运作方式相同，但是需与ex的行寻址功能和命名缓冲区一起使用。

例如：

```
:160,224ya  a
```

将第160行到第224行之间的文本拖曳（复制）到缓冲区a中。接着你可以用:e进入要放置这些文本的文件，将光标移到要放置文本的行，再输入：

```
:pu a
```

将缓冲区a的内容放置到当前所在行的下一行。

第六章

全局替换

有时在一份文档的中间，或是在草稿的结尾，你可能突然发现对某些事物的用词前后不一致。或者在使用手册中，有些从头到尾都有出现的产品名称忽然之间改名了（营销策略！）。这种事情常常发生，这时你必须把已经写好的东西从头修改，并且要修改很多地方。

要做这些修改，就需要用到功能强大的命令——全局替换（global replacement）。用一个命令，就可以自动替换文件中所有出现过的某个单词（或字符串）。

在全局替换中，ex编辑器会检查每一行中有没有某个字符模式。在包含这个模式的每一行中，ex会用新的字符串来替换这个模式。现在，我们将搜索模式当成简单的字符串，本章后面会谈到强大的模式匹配语言——正则表达式（regular expression）。

全局替换实际上会用到两个ex命令：:g（global，全局）与:s（substitute，替换）。因为全局替换的语法可能非常复杂，我们将由浅至深，一步步地往前进。

替换命令的语法如下：

```
:s/old/new/
```

它会将当前这一行中第一个出现的模式*old*改为*new*。其中的/（斜线）用来分隔命令的各部分（如果斜线位于该行的最后一个字符，则可省略）。

如下面的替换命令：

```
:s/old/new/g
```

会将当前这一行中的每一个*old*更换成*new*，并不只是更换第一个。命令：s可以在替换字符串后面加上可选项。上面例子中的g选项表示全局（g选项会影响一行中的每一个模式；不要与:g混淆，后者会影响文件中的每一行。）

在:s前面加上地址，可以将有效范围扩展到超过一行。例如，下面这个例子会将第50行到第100行之间每一个出现的*old*更改为*new*：

```
:50,100s/old/new/g
```

下面这个命令会将整个文件中的*old*改为*new*：

```
:1,$s/old/new/g
```

你也可以用%代替1,$，用于指定文件中的每一行。因此上一个命令也可以写成：

```
:%s/old/new/g
```

全局替换比起找出每一个字符串再逐一修改要快得多。因为它可以用于许多不同种类的改变，而且威力强大，故我们先介绍简单的替换，再逐步进入复杂的、与上下文相关的替换。

确认替换

在使用搜索与替换命令时，再怎么小心也不为过，但有时候得到的结果仍不能尽如人愿。你可以用u来撤销上一次的搜索与替换，只要这个命令是最近一次的编辑操作即可。但是你不可能每次都及时找出这些不正确的改变。另一个保护文件的方法是在全局替换前用:w写入文件。这至少可以让你在情况不对时回到上一次保存的结果。你也可以用:e!读出上一个版本的缓冲区内容。

聪明人在此时会小心谨慎，并且应该精确地知道文件会如何被改变。如果你想知道搜索发生的情形并在每一次替换之前做确认，可以在替换命令的结尾加上 c 选项（代表“confirm”即确认之意）：

```
:1,30s/his/the/gc
```

ex将显示找到的字符串的一整行文本，而字符串会被一串插入符号（^^^^）标记出来：

```
copyists at his school
            ^^^█
```

如果确定要替换，必须输入y（代表“yes”之意）再按下ENTER，如果不想做替换，只要按下ENTER。

```
this can be used for invitations, signs, and menus.
     ^^^█
```

将vi命令中的n（重复上一次搜索）与点号（.）（重复上一个命令）结合起来，在你

不想修改整个文件，只想对其中一部分做逐页的重复替换时，是特别有用且快速的方法。因此，假设你的主编说你在该用*that*的地方用了*which*，则你可以检查每一个出现的*which*，并且只改变不正确的部分：

/which	搜索*which*
cwthat ESC	改成*that*
n	重复搜索
n	重复搜索，省略一次改变
.	重复改变（如果应该改变的话）

与上下文相关的替换

最简单的全局替换是一个词（或短语）的替换。如果你在文件中将一个词拼错了好几次（例如*editor*拼成*editer*），可做如下的全局替换：

```
:%s/editer/editor/g
```

它会把文件中所有出现的*editer*替换成*editor*。

另外有一种稍微复杂的全局替换。这种语法让你搜索一个模式，在找到包含模式的某一行时，对另外一个字符串做替换。你可以想成是与上下文相关的替换。

语法如下：

```
:g/pattern/s/old/new/g
```

第一个g告知命令需对文件中的所有行起作用。*pattern*用于识别发生替换的行。遇到包含*pattern*的行，ex即将*old*替换成*new*。最后的g表示在哪一行中做全局替换。

例如，在本书写作时，XML指令<keycap>与</keycap>会在ESC外面加上方框，表示Escape键。你想让ESC全部采用大写，但是不想修改段落中的*Escape*，即要将所有包含<keycap>指令的*Esc*改为*ESC*，可以输入：

```
:g/<keycap>/s/Esc/ESC/g
```

如果用于搜索行的模式与用于替换的模式一样，就不必重复输入了。下面的命令：

```
:g/string/s//new/g
```

可将所有包含*string*的行中的相同字符串*string*做替换。

注意下面的命令：

```
:g/editer/s//editor/g
```

它与下一个命令意义相同：

```
:%s/editer/editor/g
```

你可以用第二种形式节省一些按键。将:g命令与:d、:mo、:co或其他ex命令结合起来也是可以的。后面会提到，你可以做全局的删除、移动与复制操作。

模式匹配的规则

在做全局替换时，像vi一类的Unix编辑器不只可以搜索固定的字符串，也可以搜索可变的模式，后者称为正则表达式（regular expression）。

当你指定了某个字符串时，搜索到的结果可能包括你不想要的结果。在文件中搜索单词的问题在于单词可能用在许多不同的地方。正则表达式可以帮助你搜索上下文中的单词。请注意正则表达式可以与vi的搜索命令/与?一起使用，当然也包括ex的:g与:s命令。

在大部分的情况下，同样的正则表达式可以在其他的Unix程序中使用，如grep、sed、awk等等（注1）。

正则表达式是将一般的字符与许多特殊的元字符（metacharacter）（注2）结合起来的表达式。元字符与其用途列于下一节。

用在搜索模式中的元字符

. *（点号）*

匹配出任何单一字符（换行符除外），请记住空格也是字符。例如，p.p匹配出字符串*pep*、*pip*或*pcp*。

*

匹配出位于此符号前的单一字符，该字符可出现零到多次。例如，bugs*匹配出*bugs*（一个*s*）或*bug*（没有*s*）。

注1：关于正则表达式，还可以参考O'Reilly出版的《sed & awk》（Dale Dougherty与Arnold Robbins合著）、《Mastering Regular Expressions》（Jeffrey E.F. Friedl著）。

注2：从技术上而言，称之为元序列（metasequence）可能更为恰当。有时候，两个字符合在一起另有特殊意义，而不再只代表某个单一字符。尽管如此，“元字符”一词常见于Unix文献中，因此我们遵循惯例使用。

*可以位于元字符后。例如，加上表示任何字符的.后，.*就表示“匹配出任何数量的任何字符”。

但有个特例:s/End.*/End/会将所有在*End*后面的字符删掉（将该行位于*Enc*后的文本替换成没有文本）。

^

当^用在正则表达式的开头时，它后面的正则表达式必须位于一行的开头。例如^Part只会匹配出位于一行开头的Part，而^...只匹配出一行的前三个字符。当不是用在正则表达式的开头时，^就只代表其本意。

$

当$用在正则表达式的结尾时，它前面的正规表达式必须位于一行的结尾，例如here:$只会匹配出位于一行结尾的here:。当不是用在正则表达式的结尾时，$就只代表其本意。

\

将其后面的特殊字符当成一般字符。例如，\.可匹配出实际的句号，而不是“任意一个字符”；*可匹配出实际的星号，而不是“任意多个单一字符”。\（反斜线）阻止特殊字符被解释为特殊意义，一般称为“字符的转义”（escaping the character）（用\\即可匹配出反斜线字符）。

[ ]

匹配出方括号里的任何一个字符。例如，[AB]匹配出*A*或*B*，而p[aeiou]t可匹配出*pat*、*pet*、*pit*、*pot*、*put*。如果匹配目标为一个范围的字符，则可用第一个字符加上连字符，再加上最后一个字符来表示。例如，[A-Z]会匹配出任何从*A*到*Z*间的大写字母，而[0-9]会匹配出任何*0*到*9*间的数字。

你可以在括号中包含两个以上的范围，也可以混合使用范围与单个的字符。例如，[:;A-Za-z()]会匹配出4种标点符号加上所有的字母。

注意： 最初开发正则表达式与vi时，只打算用于ASCII字符集。但在现今的全球化形式下，现在的系统都支持区域设置（locale），因此a到z中间有哪些字符，可能出现了不同解释。若想取得精确的结果，应该在你的正则表达式里使用POSIX方括号表达式（稍后将会讨论），并避开a-z的范围。

大部分的元字符在括号中会失去特殊意义，因此可将它们当成一般字符匹配，并不需要转义。然而，在方括号中仍然有三个元字符需要转义：\、-、]。其中-表示范围指示符，你也可以将它放在括号中的第一位。

插入符号（^）只有位于方括号中的第一个时才有特殊的意义，但与一般的 ^ 元字符意义不同。作为方括号中的第一个字母时，^表示匹配出任何一个不在方括号中字符范围内的字符。例如，[^a-z]会匹配出任何不是小写字母的字符。

\(\)

会将\(与\)间的模式保存到特殊的空间（称为“保留缓冲区”）。这种方法可以保存任意一行中的9个模式，例如，下面这个模式：

```
\(That\) or \(this\)
```

会将*That*存到保留缓冲区1中，而将*this*存到保留缓冲区2中。这些保留的模式在以后可以用\1到\9的序列重新显示。例如，要将*That or this*改成*this or That*，可以输入：

```
:%s/\(That\) or \(this\)/\2 or \1/
```

也可以在搜索或替换字符串时使用*n*表示法，例如：

```
:s/\(abcd\)\1/alphabet-soup/
```

可将*abcdabcd*换成*alphabet-soup*（注3）。

\< \>

会匹配出以某些字符开头（\<）或结尾（\>）的单词。单词的结尾与开头是由标点符号或空格来分隔的。例如，\<ac只会匹配出以ac开头的单词，如action，而ac\>只会匹配出以ac结尾的单词，如maniac；它们都不会匹配出react。请注意，这种表示法并不像 \(...\)，它不需要成对使用。

~

会匹配出任何上一次搜索时所使用的正则表达式。例如，如果你搜索过*The*，便可以用/~n来搜索*Then*。注意，它只能用在正则搜索（使用/）中（注4），而不能用在替换命令中。然而，它在替换命令中的替换部分却有类似的意义。

vi的同类品还支持其他的扩展正则表达式语法。参阅第八章的“扩展正则表达式”一节，以取得更多信息。

POSIX 方括号表达式

我们已经介绍了使用方括号来比较任意一个位于方括号中的字符，如[a-z]。POSIX 标准引进了另外的方法，用以比较非英文字母的字符。例如，法文字母“è”是一个字母

注3： 在vi、nvi、Vim中均可如此使用，但不适用于elvis或vile。

注4： 这是原始vi的一项古怪特性。在使用过后，保存的搜索模式即设为~后输入的新文本，而非预期中的新模式。由于其他的同类品都不存在这种行为，因此我们不太推荐使用。

字符，但不能用一般的字符类[a-z]匹配出来。另外，此标准也规定了一些在字符串数据匹配与校对（排序）时应该被当成一个单位的字符序列。

POSIX 也将这种技术定为标准。在POSIX标准中，方括号内的字符组称为“方括号表达式”（bracket expression）。在方括号表达式中，除了*a*、*!*等文字字符之外，还可以有其他的元素，包括：

字符类（character class）
: POSIX字符类包括了用[:与:]括起的关键字。关键字描述了不同的字符类，包括字母字符、控制字符等等（参阅表 6-1）。

校对符号（collating symbol）
: 校对符号是由多字符组成的序列，但必须被当成一个单位。使用[.与.]括起所需字符。

等价类（equivalence class）
: 等价类列出了所有应该被当成相等的字符集合，如*e*与è。它包含了区域设置中的已命名元素，用[=与=]括起来。

这三种都必须出现在方括号表达式的方括号中。例如，[[:alpha:]!]匹配出任何一个字母字符或是感叹号，而[[.ch.]]匹配出校对符号*ch*，但不等同于字母*c*或字母*h*。在法语系中，[[=e=]]可能匹配出*e*、*è*或*é*。类与匹配出的字符在表6-1中有说明。

表6-1：POSIX 字符类型

类型	比较出的字符
[:alnum:]	字母与数字字符
[:alpha:]	字母字符
[:blank:]	空格与制表符
[:cntrl:]	控制字符
[:digit:]	数字字符
[:graph:]	可打印的与可见的（不包括空格）字符
[:lower:]	小写字符
[:print:]	可打印的字符（包括空白）
[:punct:]	标点字符
[:space:]	空白字符
[:upper:]	大写字符
[:xdigit:]	十六进制数字

在 HP-UX 9.x（与较新版本）系统上的vi支持POSIX的方括号表达式。Solaris上的/usr/xpg4/bin/vi也支持（但/usr/bin/vi不支持）。nvi、elvis、Vim、vile也提供此功能。尤其是当前的GNU/Linux系统，它们在安装时对于区域（locale）的选择非常敏感，因此我们可以期待使用POSIX方括号表达式取得合理的结果，特别是只匹配大写或小写字母时。

用在替换字符串中的元字符

当你做全局替换时，前面提到的元字符只在命令的搜索部分（第一部分）中有特殊意义。

例如，当你输入：

```
:%s/1\.  Start/2.  Next, start with $100/
```

注意，替换字符串会把 . 与$当成一般字符看待，你不需要将它们转义。而假设你输入：

```
:%s/[ABC]/[abc]/g
```

如果原本希望将*A*换成*a*，将*B*换成*b*，将*C*换成*c*，你一定会很惊讶。因为在替换字符串中的方括号会被当成一般字符，所以这个命令会将所有出现的*A*、*B*、*C*换成*[abc]*这个5个字符的字符串。

要解决这个问题，需要一个方式来指定变动的替换字符串。很幸运地，还有在替换符串中有特殊意义的元字符。

n

利用\\(与\\)存储的第*n*个模式的文字做代换，*n*表示为数字1到9，而之前存储的模式（位于保留缓冲区）是从行左至行右来计算的。参阅本章第81页对\\(与\\)的解释。

\\

将后面一个特殊字符当成一般字符。反斜线在替换字符串中与搜索模式中同样是元字符。要指定匹配真正的反斜线，需输入两个反斜线（\\\\）。

&

用在替换字符串中时，会被替换成搜索模式匹配出的完整文本，这在避免重新输入文本时很有用：

```
:%s/Yazstremski/&, Carl/
```

上例的替换字符串是*Yazstremski, Carl*。&也可以替换可变的模式（用正则表达式指定的模式）。例如，要在第1行到第10行中的每一行前后加上括号，输入：

```
:1,10s/.*/(&)/
```

这个搜索模式会匹配出一整行，而&会“重现”这一行，加上你的文本。

~

与使用在搜索模式时的意义类似。找到的字符串会被最后一个替换命令中的替换文本替换，这在重复编辑时很有用。例如，你可以在一行中使用:s/thier/their/，用:s/thier/~/来重复替换另一行，而搜索模式不需要相同。

再例如，你可以在一行中使用:s/his/their/，而在另一行中使用:s/her/~/（注5）。

\u或\l

使替换字符串中的下一个字符变成大写或小写。例如，要将*yes*，*doctor*改成*Yes*，*Doctor*，可以输入：

```
:%s/yes, doctor/\uyes, \udoctor/
```

这个例子没什么意义，因为直接将字母改成大写还是比较容易的。像其他的正则表达式一样，\u与\l在可变的替换字符串中最有用。例如我们前面用过的命令：

```
:%s/\(That\) or \(this\)/\2 or \1/
```

结果是*this or That*，但是我们需要调整大小写。我们用\u将*this*（现在存储在保留缓冲区2中）的第一个字母改成大写的，用\l将*That*（现在存储在保留缓冲区1中）的第一个字母改成小写的：

```
:%s/\(That\) or \(this\)/\u\2 or \l\1/
```

结果是*This or that*（不要将数字的1与小写字母l混淆，数字1在后面）。

\U或\L与\e或\E

\U与\L和前面的\u或\l很类似，但是所有接在后面的字符都会被转换成大写或小写的，一直到替换字符串结束，或出现\e或\E为止。如果没有\e或\E，所有的替换文本都会被\U或\L所影响。例如，要将*Fortran*变成大写的，可以输入：

```
:%s/Fortran/\UFortran/
```

或是使用 & 字符来重复搜索字符串：

```
:%s/Fortran/\U&/
```

所有的模式都对大小写敏感。也就是说，搜索*the*时不会找到*The*。但你可以在模式中同时指定匹配大写与小写：

注5： 当前版本的ed编辑器将%当作替换字符串中唯一表示“上一个替换命令的替换文本”的字符。

```
/[Tt]he
```

你也可以用set ic指示vi忽略大小写。第七章还会提到其他细节。

更多替换技巧

你应该知道的关于替换命令的一些重要内容：

- 简单的:s其实与:s//~/一样。换句话说，它会重复上一次替换。当你在文件中重复相同的更改，却又不想用全局替换时，它可以节省大量的时间与按键次数。
- 如果你将&想成“同样的东西”（the same thing，也就是刚刚匹配出的内容），这个命令就比较好记了。你可以在&后加上g，让替换扩展至整行，甚至可以加上行范围：

  ☐ :%&g　　*在所有地方重复上一个替换*

- & 键可以被当成vi命令，以执行:&命令（重复上一个替换）。这比:s ENTER 省下了两次按键。
- :~命令与:&类似，但是有一点不同，它用来搜索的模式是（任何命令中使用的）上一个出现的正则表达式，而不一定是上一个替换命令中的正则表达式。

  例如（注6），在下面的命令序列中：

  ```
  :s/red/blue/
  :/green
  :~
  ```

  :~等于:s/green/blue/。

- 除了/字符，分隔字符亦可为任何非字母、非数值、非空格的字符，但是反斜线（\）、双引号（"）与竖线（|）例外。这在更改路径名称时很有用：

  ```
  :%s;/user1/tim;/home/tim;g
  ```

- 当edcompatible选项启用后，vi会记住上一次替换的标志（g表示全局，c表示确认），并继续用在下一次替换中。当你在文件中移动且希望做全局替换时，这会很有用。第一次可以这样做：

  ```
  :s/old/new/g
  :set edcompatible
  ```

  然后接下来的替换都是全局的。

  尽管名称如此，但是现在还不知道有哪一种Unix上的ed是这样做的。

注6：感谢Keith Bostic于nvi说明文档中提供了这个范例。

模式匹配的范例

除非你对正则表达式很熟悉，否则前面对特殊字符的讨论想必复杂得令人害怕。多一点范例，可能会比较清楚。在后面的范例中，方块符号（□）表示空格，而不是特殊字符。

先来看看在替换中可能用到的特殊字符。假设有一个长文件，而你想将其中所有的*child*替换成*children*。首先用:w命令保存编辑缓冲区，然后试试全局替换：

```
:%s/child/children/g
```

当你继续编辑时，发现有*childrenish*这个词，这表示你误将*childish*这个词替换掉了。用:e!回到保存的缓冲区，再试试：

```
:%s/child□/children□/g
```

注意在*child*后有空格。但是这会漏掉*child.*、*child,*、*child:* 等。在思考之后，你想起了方括号可用于匹配列表中的任一个字符，因此想到了一个方法：

```
:%s/child[□,.;:!?]/children[□,.;:!?]/g
```

上例将搜索*child*后面加上空格（□）或任一个标点符号字符（,、.、;、:、!、?）的状况。你原本希望把这些文本换成*children*加上对应的空格或标点符号，结果却变成在*children*后面拖着列出的一连串标点符号。你需要将空格与标点符号放在\(与\)中，然后可以用\1来实现重新显示。再重新试试：

```
:%s/child\([□,.;:!?]\)/children\1/g
```

当搜索到\(与\)之间的一个字符时，右边的\1会取回同样的字符。上例语法看起来复杂得可怕，但是这个命令序列可以节省你的许多工作！你花在正则表达式语法上的任何时间最后都会有千倍的回报！

然而，这个命令仍然不完美。你会发现*Fairchild*也被改变了，因此需要有一个方法来找出不是其他单词一部分的*child*。

对此，vi（但并非所有其他使用正则表达式的编辑器）有一个特殊的语法，用于表示“只有当模式是一个完整单词时才有效”。字符序列\<需要是单词的开头时进行模式匹配，而\>需要是单词的结尾才进行模式匹配。两者同时使用，即可限制单词须完整才进行模式匹配。因此，在上面的范例中，\<child\>会搜索出所有的单词*child*，不管后面接着标点符号或空格。以下是你应该使用的命令：

```
:%s/\<child\>/children/g
```

搜索一般的单词类

假设你的例程（subroutine）名称都是用*mgi*、*mgr*或 *mga*开头：

```
mgibox routine,
mgrbox routine,
mgabox routine,
```

你想保留这些前缀，但要将名称中的*box*换成*square*，则可使用下面列出的任一个命令。第一个例子展示\(与\)如何存储已找到合适文段的模式。第二个例子则显示如何搜索某个模式，但更改其他文本：

```
:g/mg\([ira]\)box/s//mg\1square/g
```

```
mgisquare routine,
mgrsquare routine,
mgasquare routine,
```

全局替换会记住是否存储了*i*、*r*、*a*字符。以此方式只在*box*为例程名称的一部分时，*box*才会被替换成*square*。

```
:g/mg[ira]box/s/box/square/g
```

```
mgisquare routine,
mgrsquare routine,
mgasquare routine,
```

与前一个命令效果一样，但是有点不安全，因为它可能更改同一行中出现的其他*box*，而不能限定为只是例程名称中的*box*。

用模式移动文本块

你也可以移动用模式作为分界的文本块。例如，假设你有一份利用troff编写、共150页的参考手册。每一页都分为三段，有三个标题：SYNTAX、DESCRIPTION与PARAMETERS。其中一页的范例如下：

```
.Rh 0 "Get status of named file" "STAT"
.Rh "SYNTAX"
.nf
integer*4 stat, retval
integer*4 status(11)
character*123 filename
...
retval = stat (filename, status)
.fi
.Rh "DESCRIPTION"
Writes the fields of a system data structure into the
status array.
```

```
These fields contain (among other
things) information about the file's location, access
privileges, owner, and time of last modification.
.Rh "PARAMETERS"
.IP "\rsfBfilename\rsfR" 15n
A character string variable or constant containing
the Unix pathname for the file whose status you want
to retrieve.
You can give the ...
```

假设决定将DESCRIPTION块移到SYNTAX块之上。利用模式匹配，你可以只用一个命令，就移动150页中的所有相关文本块！

```
:g /SYNTAX/.,/DESCRIPTION/-1 move /PARAMETERS/-1
```

这个命令的运行如下：首先，ex会寻找并标记匹配第一个模式的每一行（本例为包含*SYNTAX*的行）；接下来，逐一对标记行设置 .（点号，表示当前行），再执行命令。利用move命令，将从当前这一行（.）起到包含*PARAMETERS*的前一行（/PARAMETERS/-1）间的内容一起移到包含*DESCRIPTION*的上一行（/DESCRIPTION/-1）。

注意，ex只能把文本放置在指定行之后。要让ex将文本放在一行之前，首先必须用-1减一行，这时ex会将文本放在前一行的后面。像这样的情况，一个命令可能节省几个小时的工作（这是真有其事——我们曾经用类似的模式匹配工具重新编排达数百页的参考手册）。

用模式来定义块，同样也能在其他ex命令中运行良好。以删除所有的DESCRIPTION块为例，你可以输入：

```
:g/DESCRIPTION/,/PARAMETERS/-1d
```

这种功能强大的更动隐含在ex的行寻址语法中，甚至连有经验的用户也不一定了解。因此，当你面对一个复杂而重复的编辑工作时，应花一点时间来分析问题，试试看能否派模式匹配工具上场。

更多范例

由于学习模式匹配最好的方法就是范例，故以下列出了一些模式匹配的范例并加上解析。仔细研究这些语法，了解其工作原理，应该就可以活用这些范例了。

1. 在单词*ENTER*前后加上troff的斜体代码：

   ```
   :%s/RETURN/\\fI&\\fP/g
   ```

   请注意替换时需要两个反斜线（\\），因为在troff的斜体代码中的反斜线会被当成特殊字符。例如，只有\fI会被解释成*fI*，你必须输入\\fI才能得到*fI*。

2. 更改文件中的路径名称：

```
:%s/\/home\/tim/\/home\/linda/g
```

斜线（当作全局替换序列的分隔符号）如果是替换文本或模式的一部分时，必须用反斜线做转义，即用\/才能得到/。另一个效果相同的方法是用其他字符当作模式的分隔符号。例如，你可以用冒号当作上一个范例的分隔符号：

```
:%s:/home/tim:/home/linda:g
```

这就易读多了。

3. 将*ENTER*前后加上HTML的斜体代码：

```
:%s:ENTER:&:g
```

注意，这里用&表示实际匹配出的文本，且用冒号代替斜线作为分隔符号。

4 将第1至第10行中的所有句号改为分号：

```
:1,10s/\./;/g
```

点号在正则表达式语法中有特殊意义，必须用反斜线做转义（\.）。

5. 将所有出现的*help*（或*Help*）改为*HELP*：

```
:%s/[Hh]elp/HELP/g
```

或：

```
:%s/[Hh]elp/\U&/g
```

\U会将后面的模式改为大写的。随后的模式会是重复的搜索模式，可能是*help*或*Help*。

6. 将一个或多个空格替换成一个空格：

```
:%s/□□*/□/g
```

你应了解星号被当成特殊字符时的意义。在任何字符（或是任何匹配出一个字符的正则表达式，如.或[:lower:]]）后的星号用于找出一个或多个此字符。因此，你必须在两个空格后加上星号，来代表一个或多个空格（一个空格加上零个或多个空格）。

7. 将冒号后面的一个或多个空格换成两个空格：

```
:%s/:□□*/:□ /g
```

8. 将句号或冒号后面的一个或多个空格换成两个空格：

```
:%s/\([:.]\)□□*/\1□□/g
```

在括号中的两个字符都可用于匹配。左边的字符会被\(与\)保存到保留缓冲区，并以\1命令返回至右边。请注意括号中的特殊字符（如.）并不需要转义。

9. 将标题或单词的各种用法标准化：

```
:%s/^Note[□:s]*/Notes:□/g
```

括号中有三个字符：空格、冒号与字母 *s*。因此，模式 `Note[□s:]`会匹配出 *Note* □、*Notes*或*Note:*。后面的星号使得 *Note*（加上零个空格）与*Notes:*（正确的拼法）也会被匹配出。如果没有星号，*Note*可能会被漏掉， *Notes:*可能会被错误地改为*Notes:*□*:*。

10. 删除所有空白行：

```
:g/^$/d
```

实际上要匹配出的是以行开头（^）为开头，以行结尾（$）为结尾，中间没有内容的行。

11. 删除所有空白行以及所有只包含空格的行：

```
:g/^[□tab]*$/d
```

（本例以用*tab*表示定位符说明。）一行可能看起来是空白的，但是却包含了空格或tab（定位符）。范例10不会删除这样的行。本例如前例一样搜索一行的开始与结束，但并非找出没有内容的行，而是寻找任何数量的空格或定位符。如果没有找到空格或定位符，表示这一行为空白。要删除所有包含空白但不是没有内容的行，则该行至少需要找到一个空格或定位符：

```
:g/^[□tab][□tab]*$/d
```

12. 删除每一行开头的所有空白：

```
:%s/^□□*\(.*\)/\1/
```

用`^□□*`搜索一行开头的一个或多个空格，然后用`\(.*\)`把这一行中剩下的文本保存到第一个保留缓冲区，再用`\1`恢复这些开头没有空白的文本。

13. 删除每一行结尾的所有空白：

```
:%s/\(.*\)□□*$/\1/
```

对每一行使用`\(.*\)`以保存该行所有文本，但保存范围仅限于行末空格前。则恢复的文本中就没有结尾的空白。

本例与上例的替换对每一行只会作用一次，因此不需要在替换字符串后加上g选项。

14. 在文件中每一行的开头加上>□□：

```
:%s/^/>□□/
```

本例是将每一行的开头“替换”成>□□。当然，一行的开头（这是逻辑上的概念，不是实际的字符）不会真的被换掉！

这个命令在回复邮件或写新闻稿时很有用。用户经常会希望在回复的文字中包含原始文字的一部分。习惯上，引用的原文与回复的上下文的区别在于，引用原文的行的开头会加上右尖括号（>）以及一些空格。本例即可轻松做到这一点（一般来说，只会引用部分消息，不需要的文字可在替换前后删除）。比较先进的邮件系统会自动做这些事。然而，如果你用vi来编辑邮件，可以试试本例的命令。

15. 为后续6行的结尾加上句号：

```
:.,+5s/$/./
```

行地址表示当前这一行加上5行，$表示行的结尾。与上一个例子一样，$是一个逻辑上的概念，行的结尾不会真的被替换。

16. 逆转所有用连字符分隔部分的顺序：

```
:%s/\(.*\)□-□\(.*\)/\2□-□\1/
```

用 \(.*\)把□-□前的文本保存到第一个保留缓冲区，再用\(.*\)将剩下的部分保存到第二个保留缓冲区。接着将保存的部分恢复，再将两个保留缓冲区的内容交换。产生的效果如下：

```
more - display files
```

会变成：

```
display files - more
```

而：

```
lp - print files
```

会变成：

```
print files - lp
```

17. 将文件中的每一个单词全变成大写的：

```
:%s/.*/\U&/
```

或是：

```
:%s/./\U&/g
```

在替换字符串初始化处的\U用于告知vi把替换文本转成大写的。&字符会重新显示搜索模式匹配出的文本，作为替换字符串。本例的两个命令相同，但第一个会快很多，因为它在每一行只做一次替换（.* 可匹配出一整行，每一行一次）；而第二个会对每一行做重复的替换（. 只会匹配出一个字符，依靠最后的g做重复替换）。

18. 逆转文件中各行的次序（注6）：

```
:g/.*/mo0
```

搜索模式会匹配出所有行（包含零个或更多字符的行）。每一行会依序移动到文件的开头（即移到假想的第0行）。当匹配出的每一行移到开头时，会把前面匹配过的行逐一向下移，直到最后一行位于开头。因为所有的行都要初始化，所以可用更简洁的方法达到同样的效果：

```
:g/^/mo0
```

19. 在文本文件数据库中，对所有不包含*Paid in full*的行加上*Overdue*一词：

```
:g!/Paid□in□full/s/$/Overdue/
```

或是等效的：

```
:v/Paid□in□full/s/$/Overdue/
```

要影响除了匹配模式以外所有的行，可以在g命令前面加上 !，或者简单地用v命令。

20. 将所有不是由数字开头的行移到文件结尾：

```
:g!/^[0-9]/m$
```

或是：

```
:g/^[^0-9]/m$
```

如果^号是方括号中的第一个字符，它会逆转整个方括号内的意义，因此两个命令效果相同。第一个表示“不匹配以数字初始化的行”，而第二个表示“匹配不以数字初始化的行”。

21. 将手动编号的小标题（如1.1、1.2等等）换成troff宏（如.Ah表示A级标题）：

```
:%s/^[1-9]\.[1-9]/.Ah/
```

搜索字符串会匹配出一个非零的数字加上一个句号再加上另一个非零的数字。请注意，句号在替换字符串中不需要转义（虽然有\也不会有作用）。但这个命令找不到包含两个以上数字的章节编号。如果要做到这个功能，就要修改命令：

```
:%s/^[1-9][0-9]*\.[1-9]/.Ah/
```

现在可以匹配出第10到第99章（数字1到9后接一个数字）、第100到第999章（数字1到9后接两个数字）等等。当然这还是可以匹配出第1到第9章（数字1到9的后面不加数字）。

注6： 出自 Walter Zintz 发表于《Unix World》上的文章（1990 年5月）。

22. 将文件小标题中的数字删除。例如，想把下列范例行：

```
2.1 Introduction
10.3.8 New Functions
```

换成：

```
Introduction
New Functions
```

命令如下：

```
:%s/^[1-9][0-9]*\.[1-9][0-9.]*□//
```

搜索模式与上一个例子很像，但是数字长度不一样了。标题至少必须包含“数字、句点、数字”，因此你先尝试上一个例子中的搜索模式：

```
[1-9][0-9]*\.[1-9]
```

但是这个范例中，标题中可能还有任意数量的数字或句号：

```
[0-9.]
```

23. 将单词*Fortran*换成短语*FORTRAN*（*acronym of FORmula TRANslation*）：

```
:%s/\(For\)\(tran\)/\U\1\2\E□(acronym□of□\U\1\Emula□\U\2\Eslation)/g
```

首先，我们注意到*FORmula*与*TRANslation*都使用了原来单词的一部分，因此决定将搜索模式保存成两部分：`\(For\)`与`\(tran\)`。第一次恢复时，我们将两部分一起使用，将所有字符改为大写的：`\U\1\2`。接下来，用`\E`将大写撤销，否则接下来的替换文本也都会变成大写的。然后用实际输入的文本来替换，再恢复第一个保留缓冲区。由于这个缓冲区仍然包含*For*，因此将其首先转换成大写的：`\U\1`。紧接着，我们将剩下的部分恢复成小写的：`\Emula`。最后，恢复第二个保留缓冲区。它包含了*tran*，因此我们把前面的转换成大写的，后面的转换成小写的，再输出剩下的部分：`\U\2\Eslation`）。

模式匹配的最后叮咛

作为本章的总结，我们提出一些包含复杂模式匹配概念的示范任务。我们会一步一步来解决问题，而不是直接给出答案。

删除未知的文本块

假设你有几行文本，而一般的形式如下：

```
the best of times; the worst of times:  moving
The coolest of times; the worst of times:  moving
```

你感兴趣的行都是以*moving*结束，但是不知道开头两个单词是什么。若你想将所有以*moving*结束的行改成：

```
The greatest of times; the worst of times:  moving
```

因为这些改变必须在特定的行中，你必须指定与上下文相关的全局替换。:g/moving$/可搜索出以*moving*结尾的行。接下来，你发现搜索模式可能是任意数量的任意字符，因此想到了元字符.*，但是这会找出一整行，除非你用某些方法来限制范围。下面是第一次尝试的结果：

```
:g/moving$/s/.*of/The□greatest□of/
```

你决定的这个搜索字符串匹配出的被替换字符串为一行的开头到第一个*of*。既然需要用单词*of*来限制搜索范围，于是直接在替换字符串中重复一次*of*。以下是产生的结果：

```
The greatest of times: moving
```

有点不太对。替换的字符串把第二个*of*前的文本都去掉了，而不是第一个。为什么呢？当有选择时，“匹配任意数量的任意字符”这个动作会尽可能匹配出最多的文本。在这里，因为 *of* 出现了两次，搜索字符串会匹配出：

```
the best of times; the worst of
```

而不是：

```
the best of
```

因此，你的搜索模式必须做更多的限制：

```
:g/moving$/s/.*of times;/The greatest of times;/
```

现在的.*会匹配出任何在*of times;* 之前出现的字符。因为这只会出现一次，所以一定是第一个。

然而有时也会出现不方便使用.*元字符，甚至使用不正确的情况。例如，你可能发现要输入很多单词来限制搜索模式，或者不能用特定的单词来限制模式（如果行中的文本差异很大）。下一节会示范解决这种情况。

在文本数据库中交换条目

假设你想将（文本）数据库中所有的姓与名交换顺序。其中的数据行看起来可能像这样：

```
Name: Feld, Ray; Areas: PC, Unix; Phone: 123-4567
Name: Joy, Susan S.; Areas: Graphics; Phone: 999-3333
```

每一个字段的名称都以冒号结尾，各个字段则以分号隔开。以第一行来说，你想将*Feld, Ray*换成*Ray Feld*。我们先提出一些看起来很有希望，但实际上不能用的命令。在每个命令之后，我们会展示更改前与更改后的情况。

```
:%s/: \(.*\), \(.*\);/: \2 \1;/

Name: Feld, Ray; Areas: PC, Unix; Phone: 123-4567          更改前
Name: Unix Feld, Ray; Areas: PC; Phone: 123-4567           更改后
```

我们将第一个保留缓冲区用黑体字表示，而将第二个保留缓冲区用斜体字表示。注意第一个保留缓冲区的内容比你想象的多。因为后面的模式限制得不够，因此保留缓冲区会一直保存到第二个逗号。现在你尝试限制第一个保留缓冲区的内容：

```
:%s/: \(....\), \(.*\);/: \2 \1;/

Name: Feld, Ray; Areas: PC, Unix; Phone: 123-4567          更改前
Name: Ray; Areas: PC, Unix Feld; Phone: 123-4567           更改后
```

这里你设法将姓保存到第一个保留缓冲区，但是现在的第二个保留缓冲区保存的内容会直到这一行中的最后一个分号。你再试着对第二个保留缓冲区也做限制：

```
:%s/: \(....\), \(...\);/: \2 \1;/

Name: Feld, Ray; Areas: PC, Unix; Phone: 123-4567          更改前
Name: Ray Feld; Areas: PC, Unix; Phone: 123-4567           更改后
```

这的确是你想要的结果，但只能针对四个字母的姓与三个字母的名（前面几次尝试也有同样的错误）。为什么不回到第一次尝试，好好选一个搜索模式呢？

```
:%s/: \(.*\), \(.*\); Area/: \2 \1; Area/

Name: Feld, Ray; Areas: PC, Unix; Phone: 123-4567          更改前
Name: Ray Feld; Areas: PC, Unix; Phone: 123-4567           更改后
```

这方法有效，但我们还要继续讨论，并提出一个额外考虑。假设*Area*字段不一定存在或者不一定在第二个字段，那上一个命令又没有用了。

我们用这个问题来介绍一个概念。当你重新思考一个匹配模式时，通常比较好的做法是更精准地修改变量（元字符），而不是用特定的文字来限制模式。在模式中运用越多变量，命令的力量就会越强大。

在这个例子中，重新思考你要匹配何种模式——应该是每一个用大写字母开头，加上任意数量小写字母的单词。因此，匹配姓或名的模式可如下所示：

```
[[:upper:]][[:lower:]]*
```

姓中可能会有多个大写字母（例如*McFly*），因此你会想搜索在第二个字母之后的这种可能性：

```
[[:upper:]][[:alpha:]]*
```

这用在名中也没关系（你永远不知何时会出现*McGeorge Bundy*）。现在命令变成：

```
:%s/: \([A-Z][A-Za-z]*\), \([[:upper:]][[:alpha:]]*\);/: \2 \1;/
```

很可怕不是吗？但是上例仍然不能包含*Joy, Susan S.*这样的名字。因为名字段可能包含了中间名的首字母，你需要在第二对括号中增加一个空格与句号，但是够用就够了。有时候，精确指定理想结果比起指定不要的结果困难得多。在这个示范数据库中，姓是以逗号结尾，因此姓字段可以想成是一个不包含逗号的字符串：

```
[^,]*
```

这个模式会匹配出第一个逗号之前的字符。同样地，名字段是一个不包含分号的字符串：

```
[^;]*
```

将这些更有效率的模式用在上一个命令中，会得到：

```
:%s/: \([^,]*\), \([^;]*\);/: \2 \1;/
```

同样的命令也可作为与上下文相关的替换。如所有的行都以*Name*开头，可以用：

```
:g/^Name/s/: \([^,]*\), \([^;]*\);/: \2 \1;/
```

你也可以在第一个空格后加上星号，以匹配出冒号后有其他空格（或没有空格）的情况：

```
:g/^Name/s/: *\([^,]*\), \([^;]*\);/: \2 \1;/
```

用:g重复命令

在我们常看到的:g命令的用法上，多半是用来选择一些行，接着让后面的命令在同一行上做编辑操作。例如，用g选择某些行，再对它们做替换或删除：

```
:g/mg[ira]box/s/box/square/g
:g/^$/d
```

然而，在Walter Zintz发表于《Unix World》（注7）的论文中（分成两部分），对g命令提出了一个有趣的观点。g命令会选择某些行——但是伴随的编辑命令却不一定要影响

注7：第一部分，《vi Tips for Power Users》，发表于 1990 年4月的《Unix World》；第二部分，《Using vi to Automate Complex Edits》，发表于 1990 年5月的《Unix World》。本例出现在第二部分。

这些选择的行。

由此，他展示了一个技巧，可以重复ex命令任意多次。假设你想将当前文件的第 12 到17行复制10份并放在当前文件的结尾，你可以这样做：

```
:1,10g/^/ 12,17t$
```

这是非常意料之外的做法，但是真的可行！g命令选择了第1行，执行了指定的t命令，接着到第2行，执行下一个复制命令。当到达第10行时，ex已经复制了10次了。

收集行

以下是另一个用g命令的范例，也是来自Zintz文章中的推荐。假设你在编辑一篇有数个段落的文章，其中第二部分如下所示，我们使用 ... 表示省略的文本，并列出行号作为参考：

```
301  Part 2
302  Capability Reference
303  .LP
304  Chapter 7
305  Introduction to the Capabilities
306  This and the next three chapters ...

400  ... and a complete index at the end.
401  .LP
402  Chapter 8
403  Screen Dimensions
404  Before you can do anything useful
405  on the screen, you need to know ...

555  .LP
556  Chapter 9
557  Editing the Screen
558  This chapter discusses ...

821  .LP
822  Part 3:
823  Advanced Features
824  .LP
825  Chapter 10
```

章的编号显示在其中一行，标题在下面一行，而这一章的文字（特别标为黑体以强调）位于再下一行。你想做的第一件事，是复制每一章的初始行，送到一个现有的文件begin中。

以下是完成上述工作的命令：

```
:g /^Chapter/ .+2w >> begin
```

在使用这个命令前，光标必须位于文件的第一行。首先你搜索位于一行开头的*Chapter*，但是接下来你想对每一章的初始化行（*Chapter*往下第二行）执行命令。因为以 *Chapter*开头的行已经被选为当前行，.+2的行地址表示往下第二行。当然也可以用相应的行地址+2或++。你想将这些行写入一个名为begin的现有文件中，因此用w命令加上附加运算符>>。

假设你只想送出第二部分中每一章的开头。则需要限制用g所选择的行，因此将命令改为：

```
:/^Part 2/,/^Part 3/g /^Chapter/ .+2w >> begin
```

这里的g命令选择了以*Chapter*开始的行，但是搜索范围只限于从包含*Part 2*的行开始，到包含*Part 3*的行为止。如果你使用这个命令，文件begin的最后几行会是：

```
This and the next three chapters ...
Before you can do anything useful
This chapter discusses ...
```

这是第 7、8与 第9章开始的行。

除了这些刚刚送出的行以外，你可能想将各章的标题复制到文件的结尾，以准备制作目录。你可以用竖线添加第二个命令，如下所示：

```
:/^Part 2/,/^Part 3/g /^Chapter/ .+2w >> begin | +t$
```

凡是所有接在后面的命令，操作时的行地址均是相对于前一个命令的。第一个命令标记了某些以 *Chapter*开头的行（在第二部分中），而各章的标题出现在这些行的下一行。因此，要在第二个命令中使用各章的标题，应该使用行地址+（或等效的+1、.+1），接着再用t$将各章标题复制到文件结尾。

从这些例子中所展示的技巧可知，动脑思考与动手去做可能让你得到解决编辑问题的特殊方法。不要害怕尝试！只要先备份就不会有事！

第七章

高级编辑方法

本章介绍一些vi与ex编辑器的高级功能。在进入本章的概念介绍前，各位应该已经对前几章的内容相当熟悉了。

本章分成五个部分。第一部分讨论一些设置选项的方法，用于自定义编辑环境。我们将学到如何使用set命令以及如何用.exrc文件创建多个不同的编辑环境。

第二部分讨论如何在vi中执行Unix命令以及如何通过Unix命令用vi过滤文本。

第三部分讨论各种省下冗长命令的方法，例如使用缩写或改为只用一个按键（称为映射键）等等。另外，还提到了@功能（@-function），可以让我们把命令序列存储到缓冲区中。

第四部分讨论如何在Unix命令行或shell 脚本中使用ex脚本。脚本编码提供了强大的重复编辑途径。

第五部分讨论一些对编程者特别有用的vi功能。vi有些选项可以控制行的缩排，也有选项可以显示不可见的字符（特别是tab与换行符）。还有一些特别适用于代码块或C函数的搜索命令。

自定义vi

vi在不同的终端上有不同的运作方式。在时下的Unix系统上，vi会从terminfo 终端数据库中取得你使用的终端的运作方式（在比较旧的系统上，vi会用原始的 termcap 数据库）（注1）。

注1：　这两种数据库的位置随着厂商的不同而有所不同。可试以man terminfo与man termcap命令取得所用系统的相关信息。

还有许多可在vi中设置会影响vi运作的选项。例如，设置右边界让vi自动换行，这样就不用按回车键了。

你可以用ex命令:set在vi中改变选项。另外，只要vi被打开，就会读入位于你的主目录中的.exrc文件，以取得进一步的操作指示。在这个文件中使用:set命令，就可以改变vi的运作方式。

你也可以在本地目录中创建.exrc文件，对不同环境中使用的不同选项做初始化。例如，你可能会定义一组选项用于编辑英文文本，而另一组则用于编辑源程序。位于主目录中的.exrc会先被执行，接着才是当前目录中的.exrc文件。

任何存储在环境变量 EXINIT中的命令都会在vi打开时执行。EXINIT中的设置会比主目录中的.exrc文件先执行。

:set命令

:set命令可以改变两种类型的选项：一种是切换选项，只能选择是或否；另一种可接受数值或字符串值（如边界的位置或文件的名称）。

切换选项的默认值可能已为开或关。要将某个切换选项打开，命令是：

:set *option*

如果要将某个切换选项关闭，命令是：

:set *nooption*

例如，要指定模式的搜索要忽略大小写，则输入：

```
:set ic
```

如果要让vi回到搜索时分辨大小写的状态，则下达命令：

```
:set noic
```

有些选项需要指定某些值。例如，window选项用于设置屏幕上“窗口”所显示的行数。你可以用等号（=）来设置这些选项：

```
:set window=20
```

在vi编辑会话中，你可以检查vi正在使用的选项。命令：

```
:set all
```

会显示选项的完整列表，包含用户的设置值及vi所“选择”的默认值。

显示结果应该如下所示（注2）：

```
autoindent         nomodelines                      noshowmode
autoprint          nonumber                         noslowopen
noautowrite        nonovice                         tabstop=8
beautify           nooptimize                       taglength=0
directory=/var/tmp paragraphs=IPLPPPQPP LIpplpipnpbp tags=tags /usr/lib/tags
noedcompatible     prompt                           tagstack
errorbells         noreadonly                       term=vt102
noexrc             redraw                           noterse
flash              remap                            timeout
hardtabs=8         report=3                         ttytype=vt102
noignorecase       scroll=11                        warn
nolisp             sections=NHSHH HUuhsh+c          window=23
nolist             shell=/bin/ksh                   wrapscan
magic              shiftwidth=8                     wrapmargin=0
nomesg             showmatch                        nowriteany
```

使用以下命令，可依照名称寻找任何个别选项的当前值：

```
:set option?
```

命令:set用于显示特别更改或设置的选项，可能是在.exrc文件或当前编辑会话中有所更改的选项。

显示的结果可能如下所示：

```
number sect=AhBhChDh window=20 wrapmargin=10
```

.exrc文件

控制vi环境的.exrc文件位于你的主目录中（第一次登录时所处的目录）。你可以用vi编辑器更改.exrc文件，就像其他文本文件一样。

如果没有.exrc文件，可用vi创建。在这个文件中输入set、ab、map……这些想在使用 vi 或 ex 时产生效果的命令（ab与map将于稍后讨论）。.exrc文件样本可能如下所示：

```
set nowrapscan wrapmargin=7
set sections=SeAhBhChDh nomesg
map q :w^M:n^M
map v dwElp
ab ORA O'Reilly & Associates, Inc.
```

注2： :set all的结果与所使用的vi版本很有关系。这里的范例是一般Unix下的vi产生的结果；各种同类品所产生的结果都会不同。本表依字母顺序由上而下、由左到右排列，前面有no的选项则以拿掉no后的第一个字母排序。

因为这个文件实际上是由ex读入的（在ex进入可视模式（vi）前），所以.exrc命令前不需加上冒号。

其他环境

vi除了读入位于主目录中的.exrc，也可以读入位于当前目录中的.exrc。如此可对特别的项目设置适当的选项。

在所有现代的vi版本中，在vi读入当前目录中的.exrc文件之前，你需要先在你的主目录的.exrc文件中设置.exrc选项。

```
set exrc
```

此机制可以防止其他人在你的工作目录中放置一个其命令会危害你的系统的安全的exrc文件（注3）。

例如，你可能想在某个主要用于编程的目录中设置如下选项：

```
set number autoindent sw=4 terse
set tags=/usr/lib/tags
```

而另一个用于文本编辑的目录则用另一组选项：

```
set wrapmargin=15 ignorecase
```

请注意,可以在主目录的.exrc文件中设置某些选项，而在本地目录中取消设置的这些选项。

你也可以将选项设置保存到.exrc以外的文件并用:so命令读入（so是source的缩写），以便定义其他的vi环境。例如：

```
:so .progoptions
```

本地的.exrc文件对于定义缩写与映射键（本章稍后另有介绍）很有用。当我们编写书籍或手册时，可把书中用到的所有缩写保存于所在目录的.exrc文件中。

一些有用的选项

就像你输入:set all所见到的，可以设置的选项实在多如牛毛。其中许多是给vi内部使用的，通常不需要更改；有些在特定情况下很重要，但除此之外则未必（例如，noredraw与window对ssh会话很有用）。在第421页“Solaris的vi选项”一节的表B-1，我

注3：原始的vi版本会自动读取这两个文件，如果它们存在的话。exrc选项关闭潜在安全漏洞。

们简短地描述了每个选项，希望大家能花一点时间，试着设置这些选项。如果某个选项让你感到兴趣，试着设置（或是取消）并观察因此发生的变化。有时可能会发现令人惊奇的实用工具。

就像本书第23页“在一行中移动”小节中讨论的一样，选项wrapmargin在编辑非程序的文本时乃是基本选项。wrapmargin用于指定右边界的距离，可在输入时自动把文本换到下一行（省去手动换行的麻烦）。常见的值是7到15：

```
:set wrapmargin=10
```

另外有三个选项可控制vi搜索时的动作。正常情况下，搜索能够分辨大小写（*foo*与*Foo*不同）、回到文件开头继续搜索（表示你可以在文件中任何一处开始搜索，且仍然可以找到所有符合的文本）、于模式匹配时辨识通配符。上述三项设置的默认值分别是 noignorecase、wrapscan、magic。要更改任何一个默认值，应该设置相反的切换选项：ignorecase、nowrapscan、nomagic。

编程者可能特别感兴趣的选项包括：autoindent、showmatch、tabstop、shiftwidth、number、list，还有与之相反的切换选项。

最后，考虑使用autowrite。此设置启用时，若下达:n（下一个）命令以移到下一个要编辑的文件，用:!执行shell命令前，vi都会自动将更改过的缓冲区内容写入磁盘。

执行Unix命令

使用vi编辑时，可以显示或读入任何Unix命令所产生的结果。感叹号（!）会告诉 ex 创建一个shell，并将后续文本视为Unix命令：

```
:!command
```

因此，如果你在编辑时想检查当前的日期与时间，但又不想离开vi，可以输入：

```
:!date
```

此时时间与日期会显示在屏幕上，按ENTER即可回到原来在文件中的位置继续编辑。

如果想在一行中下达多个Unix命令，而中途不回到vi，则可以用ex命令创建一个shell：

```
:sh
```

要结束shell并回到vi时，请按CTRL-D。

:read可与Unix命令结合，把Unix命令的结果读入到文件中。一个非常简单的例子如下：

```
:r !date
```

它会将系统的日期信息读入到你的文件中。在:r命令前加上某行的地址，即可以于文件中任何一处读入命令的结果。默认情况下，会列在当前这一行的下面。

假设你正在编辑一个文件，想从名为phone的文件中读入四个电话号码并依照字母顺序排列。phone的内容如下：

```
Willing, Sue  333-4444
Walsh, Linda  555-6666
Quercia, Valerie  777-8888
Dougherty, Nancy  999-0000
```

命令：

```
:r !sort phone
```

将读入phone文件通过sort过滤后的内容：

```
Dougherty, Nancy  999-0000
Quercia, Valerie  777-8888
Walsh, Linda  555-6666
Willing, Sue  333-4444
```

假设你在编辑一个文件，想从另一个文件中找一些文本插入，但又不记得文件的名称。你可以使用花费较多时间的方法：先离开你的文件，下达ls命令，找出正确的文件名称，输入文件名，进行搜索。

或者用花费较少时间的步骤：

按键顺序	结果
:!ls	file1 file2 letter newfile practice 显示当前目录中的文件列表。发现正确的文件名称后，按ENTER继续编辑。
:r newfile	"newfile" 35 lines, 949 characters 读入新文件。

通过命令过滤文本

文本块亦可当成Unix命令的标准输入。这个命令的输出替换了缓冲区中原来的文本块。你可以在ex或vi中通过命令来过滤文本。这两种方法主要的不同在于，ex使用行地址来指示文本块，而vi是用文本对象（移动命令）来指示。

用ex过滤文本

第一个例子示范如何用ex过滤文本。假设前页范例使用的姓名不是位于名为phone的文件中，而是已经包含于当前文件中的第96至第99行。你只需输入要过滤的行地址，加上感叹号以及要执行的Unix命令。例如，这个命令：

```
:96,99!sort
```

它会将第96到第99行间的文本送给sort过滤，其结果则替换原有的内容。

用vi过滤文本

在vi中，文本通过Unix命令过滤的流程，是在任何代表文本块的vi移动命令前加上感叹号，后面再加上要执行的Unix命令。例如：

```
!)command
```

它会将下一个句子传送给*command*。

当你使用这种功能时，需要知道一些不寻常的vi反应：

- 感叹号不会立刻出现在你的屏幕上。当你键入表示要过滤的文本对象的按键时，刚才输入的感叹号才会出现在屏幕底端，但是用作引用对象的字符并不会出现。
- 文本块必须超过一行，因此只能使用移动范围超过一行的按键（G、{}、()、[[]]、+、-）。要重复这种效果，可以在感叹号或文本对象前加上数值（例如，!10+与10!+都表示下10行）。像w命令这类对象就不能运作，除非重复的数量够多，多到超过一行。你也可以使用斜线（/）加上模式与换行符来指定对象，这会将当前位置到下一个出现的模式之间的文本当成命令的输入。
- 此时的影响范围是一整行。例如你的光标位于一行中央，又下达了移动到下一个句子结尾的命令，则包含整个句子开头与结尾的所有行都会被改变，不只是这个句子本身而已（注4）。
- 有一个特殊的文本对象只能用在这种命令语法中，第二个感叹号表示当前这一行：

  ```
  !!command
  ```

  记住，不管是整个序列，还是文本对象，都可以在前面加上数值，进行重复的操作。例如，上一个例子中将第96到第99行做更动，你可以将光标移到第96行再输入：

  ```
  4!!sort
  ```

注4：　当然，凡事总有例外。使用Vim时，本例只会改变当前所在的行。

或者：

```
!4!sort
```

再看另一个例子。假设你想将文件中一部分的文本从小写的改为大写的，可以用tr命令更改这一部分的大小写。在这个例子中，第二个句子是需要通过命令来做过滤的文本块。

```
One sentence before.
With a screen editor you can scroll the page
move the cursor, delete lines, insert characters,
and more, while seeing the results of your edits
as you make them.
One sentence after.
```

按键	结果

`!)`	`One sentence after.`
	`~`
	`~`
	`~`
	`!`
	
	出现在最后一行的感叹号提示你输入Unix命令，表示要过滤的文本单位是一个句子。

`tr '[:lower:]' '[:upper:]'`	`One sentence before.`
	`WITH A SCREEN EDITOR YOU CAN SCROLL THE PAGE`
	`MOVE THE CURSOR, DELETE LINES, INSERT CHARACTERS,`
	`AND MORE, WHILE SEEING THE RESULTS OF YOUR EDITS`
	`AS YOU MAKE THEM.`
	`One sentence after.`
	
	输入Unix命令并按下ENTER。输入即被输出的文本取代了。

要重复前一个命令，语法是：

```
! object !
```

这在传送编码过的文档给nroff以将原文替换成格式化输出的情况中有时很有用（或者在编辑电子邮件时，你可能会用fmt程序将文本“美化”后再送出）。请记住，“原始”的输入已经被输出所取代了。很幸运地，如果发生错误——例如传送的文本是错误消息，而不是想要的输出时，你还可以取消命令，恢复原来的内容。

保存命令

在文件中，我们常常输入相同的冗长词组。vi与ex有许多种方法能保存一长串命令序

列，包括在命令模式或插入模式中使用的方法。当你调用执行这些已保存的序列时，只需输入几个字符（甚至一个），就可达到输入整个命令序列的效果。

单词缩写

你可以定义缩写，让vi在插入模式中时替你将输入的缩写自动展开成原文。要定义缩写，可使用ex命令：

```
:ab abbr phrase
```

*abbr*是给指定*phrase*的缩写。在插入模式中，只有在你将缩写当成单词输入时，才会展开成缩写代表的字符序列；单词内的*abbr*不会被展开。

假设在文件practice中你常常输入某些词组，像很难记的产品或公司名称。下面这个命令：

```
:ab imrc International Materials Research Center
```

会把*International Materials Research Center*缩写成*imrc*，则当你在插入模式中输入*imrc*时，*imrc*即会展开成原文。

按键顺序	结果
ithe imrc	the International Materials Research Center

缩写的展开会在按下非字母字符（例如标点）、空格、回车、ENTER（回到命令模式）时发生。当你在选择缩写时，应该选择不常在输入文本时出现的字符组合。如果你创建了一个在不该展开时会展开的缩写，可以将其取消：

```
:unab abbr
```

要列出当前所定义的缩写，应输入：

```
:ab
```

缩写所用的字符不能同时出现在所代表的词组的结尾。假设下达如下命令：

```
:ab PG This movie is rated PG
```

会得到“No tail recursion”（不能在结尾递归）的消息，而缩写也不会被设置。这个消息表示：你试着定义某个会重复展开自身的缩写，而产生无穷循环。如果命令改为：

```
:ab PG the PG rating system
```

可能会也可能不会产生无穷循环，但是这就不会产生警告消息了。例如，在System V 版

本的Unix上测试这个命令时，展开能成功执行。至于在Berkeley版本的Circa 1990上，这个缩写会重复地展开如下：

```
the the the the the ...
```

直到出现内存错误，vi结束为止。

我们在以下的vi版本中做测试，结果如下：

Solaris vi
: 结尾递归的不允许执行；缩写名称出现在展开文本中间的只会展开一次。

nvi *1.79*
: 两种都会超过内部的展开限制，因此展开动作会中止，nvi产生错误消息。

elvis、*Vim*、vile
: 两种均被检测到且只会展开一次。

如果使用 Unix 的vi或nvi，我们推荐你避免在定义的词组中重复缩写的名称。

使用map命令

当你在编辑时，可能发现自己常常使用某一组命令序列，或者有时会使用一个非常复杂的命令序列。若想省下一些按键动作或省下记忆这些序列的时间，可以用map命令将命令序列“对应”于一个没有用到的键。

map命令的作用很像ab，不同之处在于你是对vi的命令模式定义宏，而不是对插入模式定义宏。

:map x *sequence*
: 定义字符*x*映射到一个编辑命令*sequence*。

:unmap *x*
: 取消定义给*x*的编辑命令*sequence*。

:map
: 列出所有被映射的字符。

在你开始创建自己的映射字符时，必须知道不会在命令模式中使用的键才能用于用户定义的命令，如下所示：

字母
: g、K、q、V与v

控制键

^A、^K、^O、^W与^X

符号

_、*、\与=

注意： 如果设置了Lisp模式，则=会由vi使用，而好几种同类品会用它来进行文本格式化。在许多时下版本的vi中，_等效于^命令，而elvis与Vim有“可视模式”，会使用到v、V、^V等键。总而言之，必须小心测试你所使用的版本的命令模式中使用的键。

依照终端的不同，也可以将映射（map）序列关联到特殊的功能键上。

使用map，可创建简单或复杂的命令序列。举个简单的例子，你可以定义一个颠倒单词顺序的命令。在vi中，如果光标位置如下：

```
you can the scroll page
```

想把*the*放到*scroll*后的命令序列为dwelp：删除一个单词用dw；移到下个词的结尾用e；向右移动一格的命令是l；p用于置放删除的单词于新位置。保存这个序列：

```
:map v dwelp
```

即可在编辑会话中的任何时候，用一个键v颠倒两个单词的顺序。

保护按键免被ex解释

在定义映射命令时，应注意到某些键不能单纯地输入并把它们当作是命令的一部分，如ENTER、ESC、BACKSPACE、DELETE，因为这些键在ex中早已有定义了。如果想将这类按键作为命令序列的一部分，必须在前面加上CTRL-V，转换按键的正常意义。在映射命令中，按键^V表示成^字符。在^V之后的字符也不会如我们的预期。例如，换行是^M，转义是^[，退格是^H等等。

另一方面，如果要将控制字符作为映射命令，在大部分情况下，只需按着CTRL键并同时按下字母键。例如，要将^A作为映射键的方法是：

```
:map CTRL-A sequence
```

然而，有三个控制字符必须用^做转义，即为^T、^W、^X。例如对^T做映射时，必须输入：

```
:map  CTRL-V CTRL-T sequence
```

这种用法可以用在任何ex命令上，不只是map命令。这表示你可以在缩写或替换命令中输入换行符。例如，这个缩写：

```
:ab 123 one^Mtwo^Mthree
```

会展开成：

```
one
two
three
```

（此处把CTRL-V ENTER序列写作^M，也就是你的屏幕上会出现的样子。）

你也可以在全文某些特定位置加入新行。命令：

```
:g/^Section/s//As you recall, in^M&/
```

会在所有以单词*Section*开始的行开头加入包含一个词组的新行。&会恢复搜索模式。

很不幸地，有个字符在ex命令中总是有特殊意义，即使你在前面加上CTRL-V也一样。回想一下竖线（|）的特殊意义，它是多个ex命令的分隔符号。你不能在插入模式的映射键中使用竖线。

现在你知道如何使用CTRL-V在ex命令中保护某些键了，如此即可定义一些功能强大的映射序列了。

复杂的映射范例

假设你有一份名词解释，其中某一项的内容如下：

```
map - an ex command which allows you to associatea
complex command sequence with a single key.
```

你想将这份名词解释表转换成troff格式，让它看起来像这样：

```
.IP "map" 10 n
An ex command...
```

定义复杂映射最好的方法是先手动编辑，写下每个按键组合；接着用按键组合创建映射键。你要执行的步骤包括：

1. 在一行的开头插入MS宏，表示缩排的段落，并且插入第一个引号（I.IP "）。
2. 按ESC结束插入模式。
3. 移到第一个单词的结尾（e），并加入第二个引号以及空格与缩排的距离（a" 10n）。

4. 按ENTER以插入新行。

5. 按ESC结束插入模式。

6. 删除连字符以及前后的空格（3x），并将下一个单词的首字母变成大写的（~）。

如果这些编辑动作要重复很多次的话，想必很烦人。

利用:map，即可在以后用一键重现整个过程：

```
:map g I.IP "^[ea" 10n^M^[3x~
```

注意，你必须对ESC与ENTER作“引用”，也就是加上CTRL-V。^[是你键入CTRL-V加上ESC时所出现的序列，而^M是键入CTRL-V加上ENTER时所出现的序列。

现在，只要输入g，就会执行整个编辑动作。如果速度缓慢，你可以看到个别的编辑动作；如果很快的话，这就像魔术一样发生了。

如果你第一次尝试用映射键却失败，也不需要感到绝望。在定义映射键时，一点小错误就会产生非常不同的结果。输入u撤销编辑，然后再试一次。

更多映射按键的范例

接下来的范例提供一些定义映射按键时的小技巧：

1. 当你移动到单词结尾时加入文本：

   ```
   :map e ea
   ```

   大部分时候，你想移动到单词结尾的唯一理由就是加入文本。这个映射序列会自动让你进入插入模式。请注意映射的e键在vi中另有意义。你可以将vi已经使用的按键拿来作映射键，但是它原来的功能在映射键产生作用后就会消失。但在这个例子中影响不大，因为E命令通常与e的功能是相同的。

2. 对调两个单词：

   ```
   :map K dwElp
   ```

   我们在本章的前面讨论过这个序列，但是现在必须使用E（假设从这个例子开始e命令已经用于映射ea了）。回想一下，此时光标位于两个单词中的第一个。但很不幸地，由于l命令的缘故，这个序列在两个单词正好位于一行的结尾时不能使用：因为光标会先移到一行的结尾，而l不能再往右移了。以下这个方法比较好：

   ```
   :map K dwwP
   ```

   你也可以用W代替w。

3. 保存一个文件并编辑下一个文件：

```
:map q :w^M:n^M
```

请注意，你可以将按键映射到ex命令，但是要确定每一个ex命令都以回车（carriage return）结束。这个序列可以轻易地让你从一个文件移动到另一个文件，这在使用一个vi命令来打开许多个小文件时很有用。将字母q做映射，可以让你记得这个序列与“退出”（quit）类似。

4. 在单词前后加上troff的黑体代码：

```
:map v i\fB^[e\fP^[
```

这个序列假设光标位于单词的开头。首先进入插入模式，接着输入黑体字的代码。在映射命令中，你不需要输入两个反斜线来代表一个反斜线。然后回到命令模式，此时键入一个“引用”的ESC。最后在单词的结尾加上结束的troff代码，并回到命令模式。注意，当我们在单词结尾加入文本时，并不需要使用ea，因为这个序列本身已经映射到单一字母e了。这表示映射序列可以包含其他映射命令（这种使用嵌套映射序列的功能，是由vi的remap选项所控制，这个选项通常为启用状态）。

5. 在单词前后加上troff的黑体代码，即使光标不是位于单词的开头：

```
:map V lbi\fB^[e\fP^[
```

这个序列与上一个相比，除了用lb来处理将光标移到单词开头的额外工作之外，其他都相同。光标可能位于单词的中间，因此要用b将光标移到开头；但是如果光标已经位于单词的开头，则b命令会将光标移到上一个单词。要防止出现这种情况，可在用b往回移动前先输入l，使光标不会出现在单词的第一个字母。你可以定义这种序列的变形，将b改为B，或将e改为Ea。然而，l命令会让这个序列在光标位于一行结尾时不能使用（你可以加入一个空格来解决这个问题）。

6. 重复地寻找与删除一个单词或词组周围的括号（注5）：

```
:map = xf)xn
```

这个序列假设你先用/(加ENTER找到了一个左括号。

如果你选择删除这个括号，则使用map命令：用x删除这个左括号，用f)寻找右括号，再用x删除；用n重复寻找左括号的动作。

如果你不想删除括号（例如，它们并没有错误），就不要使用映射命令，而改为按n寻找下一个左括号。

你也可以修改这个映射序列，用来处理成对的其他括号。

注5： 出自Walter Zintz发表于《Unix World》1990年4月号的文章。

7. 将整行的前后加上C/C++的注释符号：

```
:map g I/* ^[A */^[
```

这个序列会在行的开头插入/*，并在行的结尾加上*/。你也可以映射替换命令，达成相同的功能：

```
:map g :s;.*;/* & */;^M
```

这里先匹配整行（用.*），并在（用&）重新显示时，在该行前后加上注释符号。注意这里使用分号作为分隔符，以免在注释中需要对/字符转义。

8. 安全地重复插入较长文本：

```
:map ^J :set wm=0^M.:set wm=10^M
```

我们在第二章中提过，在设置了wrapmargin的情况下，vi有时在重复插入较长文本时发生困难。这时用映射命令是个有用的解决方法。它暂时将wrapmargin关闭（设为0），使用重复命令后再打开wrapmargin。注意，映射序列可以同时包含ex与vi命令。

在上一个例子中，即使^J是一个vi命令（将光标往下移一行），这个映射键仍然是安全的，因为实际上它的功能与j命令相同。vi中有许多按键与其他按键的功能相同，或是极少用到。然而，你应该先熟悉vi命令，再开始大胆地定义映射键，并撤销它们的正常用途。

插入模式中的映射键

一般来说，映射只在命令模式中有用——毕竟在插入模式中的按键终究应该代表原来的意义，不应该映射到命令。然而，在map命令后加上感叹号（!），即可强制覆盖按键原来的意义，以产生插入模式中的映射行为。当你处于插入模式，但是需要暂时跳到命令模式，执行命令后再回到插入模式时，这就相当有用了。

假设你刚才输入了一个单词，但是忘了将它变成斜体（或者在前后加上引号等等），则你可以定义这种映射：

```
:map! + ^[bi<I>^[ea</I>
```

现在，于单词结尾输入+时，便会在单词前后加上HTML的斜体代码。+不会出现在文本中。

上面的序列先回到命令模式（^[），返回单词开头处加入第一个代码（bi<I>），再跳回命令模式（^[），并往前移动以加入第二个代码（ea</I>）。因为映射序列是在插入模式中开始与结束，故你可以在对单词做标示之后继续输入文本。

以下是另一个例子。假设你正在输入文本，但却发现上一行应该用冒号结束，即可以定义下列对应序列以改正（注6）：

```
:map! % ^[kA:^[jA
```

如果你在当前这一行的任何地方输入%，则会在上一行的结尾加上一个冒号。这个命令先跳回命令模式，移到上一行，再加上冒号（^[kA:）；然后回到命令模式，往下移回到原来所在的行，再回到插入模式（^[jA）。

请注意，我们在前面的映射命令中使用的是不常用的字符（%与+）。当一个字符被用于插入模式的映射键后，就不能以文本方式输入这个字符了。

要让字符恢复正常文本输入方式，使用下面这个命令：

```
:unmap! x
```

*x*表示稍早用于插入模式中作为映射键的字符（虽然在你输入时，vi会于命令行将*x*展开，看起来像取消展开文本的映射，但实际上会正确地取消字符映射键）。

插入模式的字符映射键如果能联结不会用到的特殊键与字符串就比较恰当，尤其是在可编程的功能键上特别有用。

映射功能键

许多终端都有可编程的功能键（它们在现今的位映射工作站上的终端仿真程序中会被忠实地仿真出来）。你通常可以用终端上特殊的设置模式将这些键设置成打印出某些字符。然而，用终端的设置模式对这些键进行的程序化只对这种终端有用，这也限制了一些想要自行设置功能键的程序的动作。

ex 允许使用数值来映射功能键，语法如下：

```
:map #1 commands
```

上例代表第一号功能键，以此类推（因为编辑器可以在terminfo或termcap数据库中找到对应的终端，由此可知功能键产生的转义序列（escape sequence））。

像其他键一样，映射键默认在命令模式中运用。但是若使用map!命令，就可以对一个功能键定义两个不同的值—— 一个用在命令模式，另一个用在插入模式。例如在使用HTML时，可能会想在功能键中插入字体切换的代码，如下所示：

```
:map #1 i<I>^[
```

注6： 出自Walter Zintz发表于《Unix World》1990年4月号的文章。

```
:map! #1 <I>
```

如果你在命令模式中，第一个功能键会进入插入模式，输入三个字符<I>后，再回到命令模式。如果你已经在插入模式中，则只会输入三个字符的HTML代码。

如果序列中包含了^M（表示回车），则按CTRL-M。例如，要让第一个功能键可用于映射，你所使用的终端数据库项中必须有k1的定义，如下所示：

```
k1=^A@^M
```

因此，如下定义：

```
^A@^M
```

为按下功能键后所输出的内容。

要查看功能键产生的内容，可以使用od（octal dump，八进制转储）加上-c选项（显示每个字符）。你需要在功能键之后按下ENTER，接着以CTRL-D要求od显示出信息。例如：

```
$ od -c
^[[[A
^D
0000000 033   [   [   A  \n
0000005
```

在这里，功能键送出了Escape键、两个左括号与一个A。

映射其他特殊键

许多的键盘拥有特殊键，如HOME、END、PAGE UP、PAGE DOWN等与vi命令对应的键。如果终端的terminfo或termcap描述很完整，vi就可以辨认这些键。如果并非如此，你可以用map命令让vi能使用这些键。这些键大致上会将一个转义序列（escape sequence）送到计算机中——包含一个Escape字符，加上一个或多个字符组成的字符串。为了捕捉Escape，你应该在按下特殊键前先按下^V。例如，要将IBM PC键盘上的HOME键映射到vi中具相同意义的命令，可以定义如下映射键：

```
:map CTRL-V HOME 1G
```

屏幕上会显示：

```
:map ^[[H 1G
```

类似的映射命令像这样：

```
:map CTRL-V END G                    会显示  :map ^[[Y G
:map CTRL-V PAGE UP ^F               会显示  :map ^[[V ^F
:map CTRL-V PAGE DOWN ^B             会显示  :map ^[[U ^B
```

你可能会想将这些映射键放在.exrc文件中。请注意，如果一个特殊键产生了很长的转义序列（包含许多不可打印的字符），^V只会引用开头的转义字符，而映射键也不能成功。你必须寻找整个转义序列（可能在终端机的手册上），手动将其输入，并在适当的点上作引用，而不光是按^V加上特殊键。

如果你使用不同的终端（如PC的控制台和xterm），你不能期望像那样表示的映射总能运作。因此，Vim提供了一个可移植的方式来描述这样的键映射：

```
map<Home>1G                          输入6个字符：<Home>(Vim)
```

映射多个输入键

映射多个按键并不只限于功能键，也可以对应一般的按键。这对于输入某些文本如XML或HTML时会很有用。

以下是一些:map命令，让输入XML标记更方便，要感谢Jerry Peek，他是《Leaming the Unix Operating System》的合著者（O'Reilly出版）。（以双引号开头的行是注释，稍后第123页的"ex脚本中的注释"一节中尚有讨论。）

```
" ADR: need this
:set noremap
" bold:
map! =b </emphasis>^[F<i<emphasis role="bold">
map =B i<emphasis role="bold">^[
map =b a</emphasis>^[
" Move to end of next tag:
map! =e ^[f>a
map =e f>
" footnote (tacks opening tag directly after cursor in text-input mode):
map! =f <footnote>^M<para>^M</para>^M</footnote>^[kO
" Italics ("emphasis"):
map! =i </emphasis>^[F<i<emphasis>
map =I i<emphasis>^[
map =i a</emphasis>^[

" paragraphs:
map!=P ^[jo<para>^M</para>^[O
map=P O<para>^[
map=P o</para>^[
" less-than:
map!*l&lt;
...
```

要用这些命令输入脚注时，应该先进入插入模式，再输入=f。vi会插入开始与结束的标

记，并回到插入模式，光标则位于标记中间：

```
All the world's a stage.<footnote>
<para>
█
</para>
</footnote>
```

不用说，这些宏在本书的开发过程中帮了很大的忙。

@功能

命名缓冲区提供了另一个方法来创建“宏”——一种复杂的命令序列，可以用几个按键予以重复。

如果你在文本中输入一行命令（可能是vi序列或以冒号开头的ex命令），然后将其删除，再进入某个命名缓冲区，即可用@命令执行缓冲区的内容。例如，打开一个新行，并输入：

```
cwgadfly CTRL-V ESC
```

将在屏幕上出现：

```
cwgadfly^[
```

按ESC跳出插入模式，用"gdd删除这一行并进入缓冲区g。现在当你将光标移到某个单词的开头并按下@g时，这个单词会被改成*gadfly*。

因为@会被解释成vi命令，故点号（.）也可重复整个序列，即使缓冲区中包含的是ex命令也一样。@@会重复上一个@操作，而u或U可以用来撤销@的效果。

这是一个简单的例子。@功能的用处在于可使用在非常特别的命令上。尤其是在编辑多个文件时，你可以将命令存储在命名缓冲区，并在编辑任何一个文件时拿来使用。@功能与第六章中所讨论的全局替换命令结合使用时也非常有用。

从ex执行缓冲区

你也可以在ex模式下执行保存在缓冲区中的文本。此时，你应该输入ex命令，将其删除以存入某个命名缓冲区，然后在ex的冒号提示符号下使用@命令。例如，输入以下文本：

```
ORA publishes great books.
ORA is my favorite publisher.
1,$s/ORA/O'Reilly \& Associates/g
```

光标将移到最后一行，做删除动作并进入缓冲区g："gdd。然后光标移到第一行：kk。

接着从冒号的命令行执行缓冲区中的内容：@g。此时屏幕显示如下：

```
O'Reilly & Associates publishes great books.
O'Reilly & Associates is my favorite publisher.
```

有些版本的vi在ex命令行中使用时，*与@的意义相同。另外，如果在@或*命令后面的缓冲区名称是*，则这个命令会从默认的（未命名）缓冲区取得数据。

使用ex脚本

某些ex命令只在vi中使用，例如映射、缩写等等。如果你将这些命令存储在.exrc文件，则它们会在打开vi时自动执行。任何包含可执行命令的文件被称为脚本（script）。

典型的.exrc脚本中的命令在vi之外就没有用处了。然而，你可以在脚本中存储其他的ex命令，然后在编辑一个或多个文件时执行脚本。外部脚本中大部分是替换命令。

对本书作者来说，ex脚本的一个有效应用是确保术语——甚至是拼写——在文件间保持一致。例如，假设你对两个文件执行Unix的spell命令，结果得到以下的拼写错误：

```
$ spell sect1 sect2
chmod
ditroff
myfile
thier
writeable
```

spell常常找出一些不能辨认的术语及特殊词汇，但是也发现了两个真正的拼写错误。

因为我们同时检查了两个文件，所以不知道错误是出现在哪一个文件，也不知道错误的位置。虽然有别的方法找出位置，而且这对于区区两个文件中的两个错误也不会很困难，但可以想象：如果这发生在拼写拙劣的人身上，或在一次要检查大量文件时，这个工作会变得多花时间。

要把工作简化，你可以编写一个ex脚本，其中包含以下命令：

```
%s/thier/their/g
%s/writeable/writable/g
wq
```

假设你将这几行保存到一个名为exscript的文件中。接着便可以在vi中执行这个脚本：

```
:so exscript
```

或者在Unix命令行直接将这个脚本用在文件上。然后即可如下编辑文件sect1与 sect2：

```
$ ex - sect1 < exscript
$ ex - sect2 < exscript
```

ex调用后的-s是POSIX抑制正常终端消息的方式（注7）。

如果脚本比这个例子中的长得多，就会省下许多时间。然而，你可能会好奇是否有方法避免在每次编辑文件时都重复这个过程。当然，我们可以写一个 shell script，其中包含了 ex 的调用，并且做得通用化，使得它可以用在任何数量的文件上。

shell script 中的循环

你可能知道，shell除了是命令解释器也是一种程序语言。要让ex打开多个文件，我们使用一个简单的shell script命令：for循环。for循环可以使脚本中每一个参数执行一个命令序列（对shell编程初学者来说，for循环可能是最好用的东西。即使你不写其他的shell程序，也应该记住它）。

以下是for循环的语法：

```
for variable in list
do
    command(s)
done
```

例如：

```
for file in $*
do
        ex - $file < exscript
done
```

（命令不需要作缩排，这里的缩排只是为了表示清楚。）在创建这个shell script之后，将它保存到一个名为correct的新文件中，并用chmod命令使其可执行（如果你不熟悉chmod命令以及将命令加到Unix搜索路径的步骤，可以参阅O'Reilly的《Learning the Unix Operating System》）。现在输入：

```
$ correct sect1 sect2
```

correct中的for循环会将每一个参数（由代表所有参数的$*指定列表中的每一个文件）指派给变量file，再对变量内容执行ex脚本。

用一个看得见输出的例子来了解for循环的用法可能比较容易。观察下面这个更改文件名称的脚本：

注7：ex使用负号实现这个功能。通常，为了向下兼容，两种版本均可接受。

```
for file in $
*do
      mv $file $file.x
done
```

假设这个脚本存在于一个名为move的可执行文件中，以下是我们可以做的事：

```
$ ls
ch01 ch02 ch03 move$
move ch??                          只运用在以ch开头的文件名称上
$ ls                               检查结果
ch01.x ch02.x ch03.x move
```

加一点创造力，就可把上例的脚本设计成更专门的文件更名脚本：

```
for nn in $*
do
    mv ch$nn sect$nn
done
```

此时，你要在命令行指定数值，而不是文件名：

```
$ ls
ch01 ch02 ch03 move
$ move 01 02 03
$ ls
sect01 sect02 sect03 move
```

for循环不一定将$*（所有参数）当成替换列表的值，你也可以明确指定列表，例如：

```
for variable in a b c d
```

会将*variable*依序指派给*a*、*b*、*c*、*d*，亦可替换为命令的输出结果。例如：

```
for variable in `grep -l "Alcuin" *`
```

会将*variable*依序指派成grep所找到的包含字符串*Alcuin*的每一个文件的名称（grep-l列出其文件内容匹配模式的文件名，而不会实际列出匹配模式的文件行）。

如果没有指定列表：

```
for variable
```

*variable*会被依序指派成每一个命令行参数，这有点像一开始的例子。但它实际上不等效于：

```
for variable in $*
```

而是等效于：

```
for variable in "$@"
```

两者意义有些不同。符号$*会展开成$1、$2、$3……但4个字符的序列"$@"则会展开成"$1"、"$2"、"$3"……引号会防止特殊字符被进一步地解释。

回到主题，看看我们原来的脚本：

```
for file in $*
do
    ex - $file < exscript
done
```

上例同时用到两种脚本——shell script与ex脚本，看起来可能不够优美。事实上，shell的确提供了在shell script中包含编辑脚本的方式。

here document

在shell script中，运算符<<表示将以下的行到某个字符串为止作为命令的输入（这常常称为*here document*）。用这种语法，我们可以在correct中加入编辑命令，如下所示：

```
for file in $*
do
ex - $file << end-of-script
g/thier/s//their/g
g/writeable/s//writable/g
wq
end-of-script
done
```

这里的end-of-script字符串可为任意字符串——只要不会出现在输入文本的其他地方，可用于辨别here document结束位置即可。习惯上，许多用户会用EOF或E_O_F字符串当作here document的结尾。

这些方法各有优缺点。如果你的编辑操作都只做一次且不介意每一次都重写脚本，则here document是一个有效率的方法。

然而，将编辑命令写在shell script以外的文件中更有灵活性。例如，你可以养成一个习惯，将编辑命令写在一个名为exscript的文件中，以后就只需要写一次correct 脚本即可。你可以将它存储在个人的"tools"目录中（已经加入你的搜索路径中），这样在任何时候都可以使用。

文本块排序：ex脚本样本

假设想把一份用troff编码的名词解释定义按照字母排序，且每一个术语都以.IP宏开始。另外，每一个条目的前后加上.KS/.KE宏（可保证每一个术语与其定义都会以块方

式显示出且不会跨到新的一页）。名词解释的文件如下所示：

```
.KS
.IP "TTY_ARGV" 2n
The command, specified as an argument vector,
that the TTY subwindow executes.
.KE
.KS
.IP "ICON_IMAGE" 2n
Sets or gets the remote image for icon's image.
.KE
.KS
.IP "XV_LABEL" 2n
Specifies a frame's header or an icon's label.
.KE
.KS
.IP "SERVER_SYNC" 2n
Synchronizes with the server once.
Does not set synchronous mode.
.KE
```

你可以用Unix的sort命令将文件按照字母顺序排列，但你并非想对每一行排序，而只想排列名词解释中的术语顺序，并将每一个定义与对应的术语一起移动——两者不分开。可以将每一个文本块当成一个单位，把每个块合并成一行。以下是第一版的ex脚本：

```
g/^\.KS/,/^\.KE/j
%!sort
```

每一个名词解释的条目会出现在宏.KS与.KE间。j是ex中合并行的命令（vi中等效的是J）。因此，第一个命令会将所有名词解释中的条目合成一“行”。接下来第二个命令会对文件排序，产生如下内容：

```
.KS .IP "ICON_IMAGE" 2n Sets or gets ... image.   .KE
.KS .IP "SERVER_SYNC" 2n Synchronizes with ... mode.    .KE
.KS .IP "TTY_ARGV" 2n The command, ... executes. .KE
.KS .IP "XV_LABEL" 2n Specifies a ... icon's label.  .KE
```

现在这些行均依照名词解释的条目排序了。但是很不幸地，每一行都混杂了宏与文本（我们用省略号[...]代表省略的文本）。你必须找个方法插入换行符，将各行分开。修改你的ex脚本就可以做到：在做合并前，先在文本块的合并位置做标记，再将这些标记替换为换行符。以下是修改过的ex脚本：

```
g/^\.KS/,/^\.KE/-1s/$/@@/
g/^\.KS/,/^\.KE/j
%!sort
%s/@@ /^M/g
```

前面三个命令会产生以下行：

```
.KS@@   .IP   "ICON_IMAGE" 2nn@@ Sets or gets ... image. @@ .KE
.KS@@   .IP   "SERVER_SYNC" 2nn@@ Synchronizes with ... mode. @@ .KE
.KS@@   .IP   "TTY_ARGV" 2nn@@ The ... vector, @@ that ... .@@ .KE
.KS@@   .IP   "XV_LABEL" 2nn@@ Specifies a ... icon's label. @@ .KE
```

注意@@后面的空格。这是由j命令产生的，因为它将每一个换行符转换成了一个空格。

第一个命令将原来的断行换成了@@。你不需要标记块的结尾（在.KE之后），因此第一个命令用了-1，在每个块的结尾将光标往回移一行。第四个命令会恢复断行，将标记与后面的空格替换成换行符。现在你的文件已经依照块来排序了。

ex脚本中的注释

你可能想要重复使用脚本，让它能适应新的情况。对于复杂的脚本，应该加上注释，让别人（甚至是你自己！）能够了解它的运作方式。在ex脚本中，任何双引号后的文本在执行时都会被忽略，因此双引号可以标记注释的开头。注释可以自己占有一行，也可位于任何不把引号当成命令的一部分的任何命令的结尾（例如，引号在映射命令或shell的转义序列中都有特殊意义，因此这些命令所在的行结尾不能有注释）。

除了使用注释外，也可以用命令的全名来指定命令，不过这在vi中常常花费太多时间。最后，如果你再加上空格，前面一节的ex脚本会更容易阅读：

```
" 标记在每一个 KS/KE 块间的行
 global /^\.KS/,/^\.KE/-1 s /$/@@/
" 现在将块合并成一行
global /^\.KS/,/^\.KE/ join
" 将每个块排序——现在实际上各占一行
%!sort" 将合并的行恢复成原来的块
% s /@@ /^M/g
```

令人惊讶的是，substitute命令在ex中不能使用，而其他命令的全名则可以使用。

在ex之外

如果这些讨论唤起了追求更强大编辑功能的欲望，你应该注意到Unix提供了比ex更有威力的编辑器：sed流编辑器与awk数据操纵语言，当然还有非常热门的perl程序语言。要了解更多有关的信息，可以参阅 O'Reilly 的《sed & awk》、《Perl 语言入门》与《Programming Perl》等书籍。

编辑程序源代码

到目前为止，我们所讨论的功能都围绕在编辑常规文本或程序源代码上。然而，其还有一些主要是编程者会感兴趣的额外功能，包括缩排控制、搜索过程(procedure）的开始

与结束以及ctags的使用。

以下的讨论出自Mortice Kern Systems所提供的说明文档。他们提供了质量极佳、可在DOS 与Windows系统上使用的vi，可作为MKS Toolkit中的一部分。以下转载已经过Mortice Kern Systems的同意。

缩排控制

程序的源代码与一般的文本有许多不同。其中一个最重要的差异在于，代码要使用缩排。缩排显示了程序的逻辑结构，也就是各个语句组合成块的方式。vi提供了自动缩排控制，如果要使用，可下达这个命令：

```
:set autoindent
```

现在，当你用空格或tab做缩排时，后面的行会自动以相同的距离缩排。当你结束第一个有缩排的行后按下ENTER时，光标会移到下一行，并自动与上一行缩排相同的距离。

编程者将发现，这可以节省许多缩排工作的时间，尤其是在有多层缩排的情况下。

当你在自动缩排的情况下输入代码时，在一行开头按下CTRL-T会出现另一层的缩排，而按下CTRL-D则可回到上一个缩排的层级。

我们必须指出的是，CTRL-T与CTRL-D是在插入模式中输入的，与其他在命令模式中输入的命令大不相同。

该命令还有两种变形（注8）：

^ ^D

当你输入^ ^D（^ CTRL-D）时，vi会将光标移回当前这一行的开头，但是只有当前这一行。进入下一行时，仍然会从当前的自动缩排层级开始。这在 C/C++ 代码中输入 C 的预处理器命令时特别有用。

0 ^D

当你输入 0 ^D 时，vi会将光标移回当前这一行的开头。另外，当前的自动缩排层级会复位为零。进入下一行时，就不会再做自动缩排了（注9）！

当你在输入源代码时，可试着使用autoindent选项。它会简化做出正确缩排的工作，甚至可以避免错误——例如在C代码中，常常需要在回到上一层的缩排时，加上表示缩排

注8：此处的命令变形不适用于elvis。

注9：nvi 1.79版说明文档对这两个命令的说明与我们相反，但程序实际上仍照此处的说明方式运行。

结束的花括号（}）。

<<与>>命令在对代码做缩排时帮助也很大。>>默认往右移动8个空格（加入8个空格的缩排），而<<是向左移动8个空格。例如，将光标移到一行的开头并按下>键两次（>>），你会看到整行都往右移了。如果你现在按下<<<键两次，整行会再往左移。

你可以移动多行，只要在 >>或<<前加上数值就可以了。例如，将光标移到一段文本的开头，然后输入5>>，这会将此段文本的前5行往右移动。

上述的默认移动是8个空格（往左或往右）。默认值可以在命令行做改变：

```
:set shiftwidth=4
```

你将发现，让shiftwidth值与制表位（tab stop）的距离相同会很方便。

vi在做缩排时比较聪明。通常，当你看到文字一次缩排8个空格时，vi会自动在文件中加入tab字符，因为tab通常展开成8个字符。这是Unix的默认值，最容易注意到的地方是，当你在做一般输入时输入了一个tab，然后此文件被送到打印机时，Unix会将tab展开成8个空格。

如果你希望的话，可以用tabstop选项更改vi在屏幕上显示tab的方式。例如有些文本缩排太深时，你或许想修改tab成展开为4个字符，让这些行不会移到下一行去。以下的命令可达成此效果：

```
:set tabstop=4
```

注意： 我们并不推荐更改制表位。虽然vi在显示文件时可以用设置为任何长度的tab，但是在所有其他的Unix程序中，文件中的tab字符仍然会展开成8个字符。

还有更糟的事：混用tab、空格与不常用的制表位，将使你的文件在此编辑器外完全不能阅读，例如使用more之类的pager或打印时。8个字符宽的制表位是Unix中的一项规定，你必须习惯它。

有时缩排会出现意料之外的结果，因为你认为的tab字符实际上是一个或多个空格。一般来说，你的屏幕会将tab与空格都用空白来代替，不能加以分辨。然而

```
:set list
```

这个命令可以改变你的显示状态，让tab看起来是控制字符^I，并让行结尾看起来是$。如此即可认出真正的空格，也可以知道行结尾是否有多余的空格。有一个作用相同的临时命令是:l。例如：

```
:5,20 l
```

会显示第5行到第20行的tab字符与行结束字符。

一个特殊的搜索命令

(、[、{、<都可以称为开括号(opening bracket)。当光标位于任何一种开括号上时，按下%键，可将光标往前移到成对的闭括号(closing bracke)——)、]、}、>）上，程序还会同时注意一般的嵌套括号规则（注10）。例如，如果你将光标移动到下面的第一个(上：

```
if ( cos(a[i]) > sin(b[i]+c[i]) )
{
      printf("cos and sin equal!\n");
}
```

按下%，会看到光标移到这一行结尾的圆括号上。这是对应的闭括号。

同样地，如果光标位于某一个闭括号上，此时按%，光标会回移到对应的开括号上。例如，将光标移到printf这一行最后的括号上，再按%观察。

vi甚至可以帮你寻找括号。如果光标不是位于括号字符上，当你按下%时，vi会在当前这一行中往前寻找第一个开或闭括号，再将光标移到对应的括号上！以前例而言，如果光标位于第一行的>上，%会找到第一个开括号，并将光标移到对应的闭括号上。

这个搜索字符不只能帮你在程序中大幅往前或往后移动，还可以检验程序中的嵌套括号。例如将光标放在C函数的第一个{上，按%应该会移到（你认为的）函数结尾。如果不是的话，就出现问题了。如果文件中没有对应的}时，vi会发出蜂鸣声。

另一个寻找对应括号的技巧是打开以下这个选项：

```
:set showmatch
```

它不像%，设置showmatch（或是其缩写sm）后，可在插入模式中提供帮助。当你输入)或}时（注11），光标会先暂时移动到对应的(或{处，再回到当前的位置。如果对应者不存在的话，终端即发出蜂鸣声。如果对应者在屏幕之外，vi会继续寻找下去。

使用标签（tag）

一个庞大的C或C++源代码经常分成许多个文件。有时候，实在很难跟踪哪一个文件中有哪些函数。对此，Unix中有一个ctags命令，可以与vi的:tag命令一起使用。

注10：在我们测试的版本中，只有nvi支持用%对应<与>。vile有选项能够设置可用%对应的字符组合。

注11：在elvis、Vim、vile中，showmatch也会显示成对的方括号（[与]）。

注意： Unix版本的ctags会处理C语言，通常也会处理Pascal与Fortran 77，有时甚至可以处理汇编语言；然而，它几乎不会处理C++。有某些版本可以产生C++或其他语言及文件类型所使用的标签文件，请参考第132页的“增强的标签”一节。

ctags命令在Unix命令行上执行。它的功能是产生一个信息文件，让vi可用该文件判断各个文件中分别定义了哪些函数。默认情况下，这个文件名为tags。在vi中，下面这样的命令：

```
:!ctags file.c
```

会在当前目录中创建一个名为tags的文件，其中包含file.c中定义的所有函数的信息。而下面这个命令：

```
:!ctags *.c
```

会创建描述所有C源代码文件的tags文件。

现在假设你的tags文件包含了创建某个C程序所需的全部源文件的信息，再假设你想观察或编辑程序中的某个函数，但是不知道确切的位置。则在vi中，这个命令：

:tag *name*

会在tags文件中寻找哪一个文件包含了函数*name*的定义。接着vi读入这个文件，并将光标指到定义*name*函数的那一行。这样你就不需要知道要编辑的文件的名称，只要知道想编辑的函数即可。

你也可以在vi的命令模式中使用标签的功能。将光标移到你要查找的标识符上，再输入^]。vi会执行标签查找，并将光标移到定义了标识符的文件中。在移动光标时要小心，vi 会使用以当前光标所在位置为起点的“单词”，而不是包含光标的那个完整单词。

注意： 如果你尝试使用:tag命令读入一个新文件，但当前的文件在上一次修改后还没保存，vi便不会让你进入新文件。你必须先用:w写入当前的文件，才能使用:tag。亦可输入：

:tag! *name*

强迫vi放弃编辑结果。

Solaris版本的vi实际上支持标签栈（stack）。然而，在Solaris的联机说明中似乎完全没有提到。因为Unix上大部分版本的vi都不支持标签栈，故我们将这个特性移到第八章的“标签栈”一节特别介绍。

第八章

vi同类品的功能总览

它们都是我兄弟

有许多可以自由取得的vi编辑器同类品。附录D提供了一个网站，列出了所有已知的vi同类品;第二部分的所有篇章则将详细介绍Vim；第三部分将再介绍另外三种常用的同类品，包括：

- Keith Bostic 的 nvi 1.79 版（第十六章）
- Steve Kirkendall 的 elvis 2.2.0 版（第十七章）
- Kevin Buettner、Tom Dickey与Paul Fox的vile 7.4版（第十八章）

这些同类品会出现的原因在于vi的源代码不能自由取得——因此不能将vi移植到非Unix的环境中，或研究它的代码。另外也是因为Unix的vi（或是其他的同类品！）没有提供某些功能。例如，Unix的vi通常限制一行的最大长度，而且也不能编辑二进制文件（在后面谈到各种程序时，会提供更多的历史信息）。

每一种程序都提供了Unix的vi之外的许多扩展功能。尽管方式可能不同，但许多同类品通常会提供相同的扩展功能。我们不在各个程序的章节重复这些共同特性，而在这一章中一次解决。你可以将这一章当成展示“同类品能做什么事”，而将它们各自的章节当成展示“同类品怎么做这些事”。

本章讨论功能主题的顺序也将用在第二部分讨论Vim的章节上，同时也将以较为浓缩的方式运用于第三部分。本章内容包括：

多窗口编辑

可将屏幕分割成多个“窗口”，另外或许加上在图形用户界面（GUI）环境中使用多窗口的功能。你可以在每个窗口编辑不同的文件，或观察同一文件的不同部分。这项功能超越正统的vi，或许也是最重要的一个扩展功能。

图形用户界面

除了nvi外，所有的同类品都可以编译成支持X Window界面。如果系统上运行X，此时采取图形用户界面（GUI）的版本，可能会比采取分割xterm（或其他终端仿真器）的屏幕更好用。图形用户界面的版本通常提供滚动条与多种字体等便利功能，其他操作系统的原生图形用户界面可能也会支持这些功能。

扩展正则表达式

所有vi同类品都可以使用与Unix的egrep命令相似或相等的正则表达式来做文本匹配。

增强的标签

如第126页中“使用标签”一节所提到的，你可以用ctags程序对文件创建一个可搜索的数据库。通过在你做标签搜索时将当前的位置存储起来，这些同类品能够支持“栈”标签，以后你就可以回到这个位置。如果有多个位置，可以用后进先出（LIFO）的顺序来存储，产生一个关于位置的栈。

许多vi同类品的作者以及至少一种ctags同类品的作者，共同针对ctags格式的增强版本定义了标准格式，尤其是对于允许函数名称过载（overload）的C++程序，使用标签功能也变得容易得多了。

改进的编辑工具

所有的同类品都提供了编辑ex命令行、不限次数撤销、任意长度的行与八位数据、增量搜索、将屏幕从左往右滚动以代替绕排过长的行以及模式指示器等其他特性。

编程辅助

许多编辑器都提供在典型的“编辑—编译—调试”的软件开发周期中完全不需要离开编辑器的功能。

语法高亮显示

在elvis、Vim、vile中，你可以设置将文件中的不同部分用不同的颜色与字体来显示。这在编辑程序源代码时特别有用。

多窗口编辑

相对于标准vi，也许各种同类品提供的最重要功能就是能在多个窗口中编辑了。这使得同时在多个文件中工作更加容易，在文件间剪贴文本也更加方便。

注意： 在同类品中，不需为了跨文件的拖动与放置而分割屏幕，只有原始的vi 才会在切换文件时丢弃存储有剪贴内容的缓冲区。

在每一种编辑器的多窗口实现下，有两个基础概念：缓冲区与窗口。

缓冲区（buffer）保存了要编辑的文本。这些文本可能来自文件，也可能是最后必须写入文件的新文本。任何一个文件都只有一个对应的缓冲区。

窗口（window）提供对缓冲区的观察窗口，让我们可以查看与修改缓冲区中的内容。同一个缓冲区可能有多个对应的窗口。在一个窗口中对缓冲区所做的更动会在该缓冲区的其他打开窗口中显现出来。缓冲区也可能没有对应的窗口。在这种情况下，你能对缓冲区做的事并不多，虽然可以针对它开一个对应的窗口。关闭对应到某个缓冲区的最后一个窗口，可以有效地“隐藏”文件。如果缓冲区被改变但还没写入磁盘，则编辑器可能会也可能不会让你关闭对应缓冲区的最后一个窗口。

当你创建新的窗口时，编辑器会分割当前的屏幕。对大部分的编辑器来说，新窗口显示编辑中的文件。接着我们切换到准备用于编辑下一个文件的窗口，指示编辑器打开并编辑新的文件。每一种编辑器都提供了vi与ex命令，可用于窗口间切换、更改窗口尺寸、隐藏与显示窗口。

第十一章专门讨论了Vim中的多窗口编辑。在第三部分的其他编辑器的相关章节中，我们展示了一个样本分割屏幕（编辑两个相同文件），并描述了如何分割屏幕以及在窗口间移动。

图形用户界面

elvis、Vim与vile也提供了图形用户界面的版本，可以利用位映射显示器（bitmapped display）以及鼠标。除了支持Unix下的X Windows之外，也可以支持Microsoft Windows或其他窗口系统。表8-1整理了几种同类品能够使用的图形用户界面。

表8-1：可用的图形用户界面

编辑器	Terminal	X11	Microsoft Windows	OS/2	BeOS	Macintosh	Amiga	QNX	OpenVMS
vim	•	•	•	•	•	•	•		
nvi	•								
elvis	•	•	•	•					
vile	•	•	•	•	•			•	•

扩展正则表达式

可在vi的搜索与替换正则表达式中使用的元字符已在第六章“用搜索模式中的元字符”一节介绍，请见第78页。每一种同类品都提供了某种形式的扩展正则表达式，有些可能需要设置某些选项，有些随时都可以使用。一般来说它们都与egrep相同（或几乎相同）。不幸的是，每一种扩展方式的“风味”都与众不同。

为了让各位感受扩展正则表达式的功能，我们以nvi举例。Vim有自己的“扩展的正则表达式”章节（第173页），而其他vi同类品的扩展语法列于第三部分，以不重复范例为原则。

nvi的扩展正则表达式是由POSIX标准所定义的Extended Regular Expression（ERE）。要启用这项特性，需要在.nexrc文件或是ex冒号提示符下使用set extended选项。

除了第六章元绍的标准元字符以及“POSIX方括号表达式”一节（第81页）中提到的表示法外，其还可以使用以下元字符：

|

表示交替（alternation）。例如，a|b表示匹配出*a*或*b*即可。然而，并不是只能使用一个字符：house|home可匹配字符串*house*或*home*。

(...)

表示集合（group），可供其他的正则表达式运算符来使用。例如，house|home可以缩写成ho(use|me)。运算符*可运用于括号中的文本：如(house|home)*可匹配出*home*、*homehouse*、*househomehousehouse*……依此类推。

当extended被设置时，用圆括号括起来的文本，就像在一般的vi中用\(...\)括起来的文本一样。实际的匹配文段（matches）可以在替换命令中的替换部分由\1，\2……取用。在这里，\(实际表示左圆括号字符。

+

+前面的模式需出现一或多次。模式可能是一个字符或是一组由圆括号括起来的字符。要注意+与*的不同。*允许匹配不成功，但是+的匹配至少必须成功一次。例如，ho(use|me)*能匹配出*ho*和*home*和*house*，但是ho(use|me)+不会匹配出*ho*。

?

? 前面的模式需出现零或一次。这表示模式为“可选”文本，可以出现也可以不出现。例如，free?d会匹配出*fred*或*freed*，但无其他结果。

{...}

定义一个区间表达式（interval expression）。区间表达式描述了指定数量的重复。下面提到的*n*与*m*是表示整数的常量：

{*n*}

刚好匹配出*n*个指定的正则表达式。例如，(home|house){2}将匹配出*homehome*、*homehouse*、*househome*、*househouse*，其他都不符合匹配条件。

{*n*,}

指定的正则表达式应匹配*n*次以上，可以想成“至少*n*次”的重复。

{*n*,*m*}

相似文段重复次数为*n*到*m*次。此时的边界条件很重要，因为它控制了在替换命令执行时将替换多少文本（注1）。

当extended没有设置时，nvi的\{与\}提供了相同的功能。

增强的标签

“Exuberant ctags”程序是ctags的同类品，比Unix上的ctags功能更强大。它可以产生扩展的tags文件格式，让标签的搜索与匹配处理更灵活与强有力。我们先介绍Exuberant版，因为许多种vi同类品都支持它。

这一节也会介绍标签栈：用:tag或^]命令来存储多个访问过的位置。所有的同类品都支持标签堆栈(tag stacking)。

Exuberant ctags

“Exuberant ctags”程序由Darren Hiebert设计，其主页位于*http://ctags.sourceforge.net/*。在本书撰写时，版本是5.7。下面的程序特色列表从ctags程序发布版本的README文件中节录：

- 能够产生所有类型的C与C++语言标签，包括类型名称、宏定义、枚举名称、枚举器（枚举模式内的值）、函数（或方法）定义、函数（或方法）原型／声明、结构成员与类数据成员、struct名称、typedef、union名称与变量。
- 支持C与C++代码。
- 另外还支持29种程序语言，包括C#与Java。

注1：*、+、?运算符分别可解析为{0,}、{1,}、{0,1}，但前者运用较方便。另外，在Unix的正则表达式开发史上，区间表达式出现的比较晚。

- 在解析代码时非常可靠，并且很不容易被包含#if预处理器条件构造的代码所愚弄。
- 列出供人阅读用的源文件中的被选择对象的列表。
- 支持产生GNU Emacs样式的标签文件（etags）。
- 可以在Amiga、Cray、MS-DOS、Macintosh、OS/2、QDOS、QNX、RISC OS、Unix、VMS与Windows 95至XP上运作。可于它的网站上取得预编译的二进制文件。

接下来将介绍 Exuberant ctags所提供的tags文件格式。

新的tags文件格式

传统的tags文件有三个以制表符分隔的字段：标签名称（一般是标识符）、包含标签的源文件以及何处可找到标识符的指示符。指示符可以是简单的行编号，或是以斜线或问号括起的nomagic搜索模式。另外，tags文件永远是经过排序的。

这是由Unix的ctags程序所产生的格式。事实上，许多版本的vi都允许搜索模式字段中出现任何的命令（一个很大的安全漏洞）。还有，由于一个未公开的实现方式，使得一行结尾若为分号接着双引号（;"）时，则这两个字符后面的文本都会被忽略（因为双引号表示注释的开头，这与在.exrc文件中一样）。

新的格式向下兼容于传统格式。前面三个字段仍然相同：标签、文件名称与搜索模式。Exuberant ctags只会产生搜索模式，而不是任意命令。扩展的属性会放在用于分隔的;"字符后。属性间以tab字符分隔，每一个属性均由以冒号分隔的两个子字段组成——第一个子字段是描述属性的关键字，第二个则是实际的值。表8-2列出了支持的关键字。

表 8-2：扩展的ctags关键字

关键字	意义
kind	其值为单一字母，表示标签的词汇类型。可用f表示函数，v表示变量等等。由于默认的属性名是kind，因此可用单一字母来表示标签的类型（例如用f代表函数）
file	这是针对“静态”的标签，也就是在本地文件中的标签。其值应该是文件的名称。 如果其值是空字符串（只有file:），则会以文件名字段为准。加上这个特例，一方面是为了让结构更紧凑，另一方面则是提供一个简单的方法，以处理不在当前目录中的tags文件。文件名字段的值永远是相对于tags文件本身所处的目录
function	针对本地的标签。其值是所定义的函数的名称
struct	针对struct中的字段。其值是结构的名称

表 8-2：扩展的ctags关键字（续）

关键字	意义
enum	针对enum数据类型中的值。其值是enum类型的名称
class	针对C++的成员函数与变量。其值是类的名称
scope	大部分针对C++的类成员函数。对私有成员，其值通常是private；对公有成员，则通常忽略其值。因此用户可以限制标签只搜索公有成员
arity	针对函数。定义参数的个数

如果字段中没有冒号，则会假定是kind类型。以下举些例子：

```
ARRAYMAXED      awk.h    427;"   d
AVG_CHAIN_MAX   array.c   38;"   d     file:
array.c         array.c    1;"   F
```

ARRAYMAXED是一个C的#define宏，定义在awk.h中。AVG_CHAIN_MAX也是一个C的宏，但是只在array.c中使用。第三行有点不一样，它是实际源文件的标签！这是Exuberant ctags使用-i F选项产生的结果，可以让你使用:tag array.c命令。更有用的是，你可以将光标移到文件名上，再使用^]命令移到该文件中（例如你在编辑 Makefile，然后希望跳到特定源文件）。

在每一个属性代表值的部分中，必须将反斜线、制表符（tab）、回车（carriage return）与换行（newline）字符分别编码成\\、\t、\r与\n。

扩展的tags文件中可能有某些以!_TAG_开头的初始标签。这些标签通常会位于文件的前面，对于辨别创建文件的程序很有帮助。以下是Exuberant ctags产生的结果：

```
!_TAG_FILE_FORMAT       2   /extended format; ..../
!_TAG_FILE_SORTED       1   /0=unsorted, 1=sorted/
!_TAG_PROGRAM_AUTHOR    Darren Hiebert   /darren@hiebert.com/
!_TAG_PROGRAM_NAME      Exuberant Ctags  //
!_TAG_PROGRAM_URL       http://home.hiwaay.net/~darren/ctags   /.../
!_TAG_PROGRAM_VERSION   2.0.3            /with C++ support/
```

编辑器可能会利用这些特殊标签而实现特殊功能。例如，Vim会注意到!_TAG_FILE_SORTED这个标签，如果文件真的排序过，即可使用二元搜索代替线性搜索来搜索tags文件。

如果你使用tags文件，我们推荐你取得Exuberant ctags并安装使用。

标签栈

ex的:tag命令与vi模式的^]命令，根据tags文件提供的信息，提供了能力有限的标识符

寻找方式。而每一种同类品都扩充了维护标签位置栈（stack）的功能。每一次下达 ex 的 tag命令或于vi模式中使用^]命令时，编辑器会在搜索指定的标签前先保存当前的位置。这样以后你就可以用vi模式下的^T命令或用ex命令回到保存起来的位置。

下面展示一个Solaris vi标签堆栈(tag stacking)的范例。Vim的标签堆栈将于第269页的“标签堆栈”小节里说明。其他同类品处理标签堆栈的方式则于第三部分的各自章节中介绍。

Solaris vi

令人惊奇的是，Solaris版的vi支持标签堆栈。而这个特色在Solaris的ex(1)与vi(1)说明手册中完全没有提到。为了完整起见，我们利用表8-3、表8-4与表8-5，汇总了Solaris 版vi的标签堆栈功能。Solaris vi中的标签堆栈非常简单（注2）。

表8-3：Solaris vi标签命令

命令	功能
ta[g][!] *tagstring*	依照tags文件中的定义，编辑包含*tagstring*的文件。如果当前的缓冲区被改变了但是还没有保存，!会强迫vi切换到新文件
po[p][!]	弹出标签栈中的一个元素

表8-4：Solaris vi命令模式中的标签命令

命令	功能
^]	在tags文件中寻找光标所在位置的标识符，并移到标识符的位置。如果启用标签堆栈，当前的位置会自动压入标签栈
^T	回到标签栈的前一个位置，即弹出一个元素

表 8-5：Solaris vi的标签管理选项

选项	功能
taglength, tl	控制要寻找的标签中的有效字符的数目。默认值为零，表示所有字符都是有效字符
tags, tagpath	此选项值是用来寻找标签的文件名称列表。默认值是"tags/usr/lib/tags"
tagstack	设为true时，vi会将每个位置放到标签栈中。利用:setotagstack来禁用标签栈

注2：这些信息根据实验而来，各人情况或有不同（your mileage may vary，YMMV）。

Exuberant ctags与Vim

为了让大家感受何为标签栈，我们利用Exuberant ctags与Vim来介绍一个简短的例子。

假设你在编写一个用到GNU的getopt_long函数的程序，且需要对它多一点了解。

GNU的getopt包含了三个文件：getopt.h、getopt.c与getopt1.c。

首先，创建tags文件，然后开始编辑主程序，其位于main.c：

```
$ ctags *.[ch]
$ ls
Makefile   getopt.c   getopt.h   getopt1.c   main.c   tags
$ vim main.c
```

按键顺序	结果

/getopt	`/* option processing. ready, set, go! */`
	`for (optopt = 0, old_optind = 1;`
	`     (c = getopt_long(argc, argv, optlist, optab, NULL)) != EOF;`
	`     optopt = 0, old_optind = optind) {`
	`       if (do_posix)`
	`               opterr = TRUE;`
	
	编辑main.c并移到调用getopt_long的地方。

^]	`int`
	`getopt_long (int argc, char *const *argv, const char *options,`
	`              const struct option *long_options, int *opt_index)`
	`{`
	`  return _getopt_internal (argc, argv, options, long_options, opt_index, 0);`
	`}`
	
	`"getopt1.c" 192L, 4781C`
	
	对getopt_long做标签寻找，Vim会移动到getopt1.c，并将光标移到getopt_long的定义上。

此时会显示出getopt_long是_getopt_internal的“包装”函数（wrapper）。再将光标移到_getopt_internal上并做另一次标签寻找。

按键顺序	结果

3jf_ ^]	`int`
	`_getopt_internal (int argc, char *const *argv, const char *optstring,`
	`                  const struct option *longopts, int *longind, int long_only)`
	`{`
	`  int result;`
	
	`  getopt_data.optind = optind;`
	`  getopt_data.opterr = opterr;`

按键顺序	结果
	(screen shown below)

```
  result = _getopt_internal_r (argc, argv, optstring, longopts,
                               longind, long_only, &getopt_data);

  optind = getopt_data.optind;
"getopt.c" 1225L, 33298C
```

现在你已经移动到getopt.c了。要寻找更多关于struct option的信息，请将光标移到option上，再做一次标签寻找。

jfo; ^]

```
  one). For long options that have a zero `flag' field, `getopt'
  returns the contents of the `val' field. */

struct option
{
  const char *name;
  /* has_arg can't be an enum because some compilers complain about
     type mismatches in all the code that assumes it is an int. */
  int has_arg;
  int *flag;
  int val;
};

/* Names for the values of the `has_arg' field of `struct option'.   */

"getopt.h" 177L, 6130C
```

编辑器现在移到getopt.h中struct option的定义上了。你可以查看解释其用途的注释。

:tags

```
 # TO tag         FROM line in file/text
 1 1 getopt_long       310 main.c
 2 1 _getopt_internal 67 getopt1.c
 3 1 option           1129 getopt.c
```

Vim中的:tags命令显示标签栈。

输入^T三次，可移回main.c中寻找开始的地方。标签功能让你在编辑源代码时更加得心应手。

改进的便利功能

所有同类品都提供了额外功能，让简单的文本编辑更容易、更有威力。

编辑ex命令行

在你输入ex命令时能够编辑ex模式命令，可能还包括编辑ex命令的历史记录。另外，有自动补全文件名称的功能，可能还包括补全命令或选项等等。

没有行长度的限制

基本上能够编辑任意长度的行。另外，还能够编辑包含任何8位字符的行。

不限次数的撤销

可以成功地连续撤销所有对文件所做的改变。

增量搜索

输入欲搜索模式时可立即在文本中开始搜索的功能。

左/右滚动

让过长的行超出屏幕的边缘，而不是绕成下一行。

可视模式

选择任意连续块的文本供后续操作之用。

模式指示器

显示插入模式与命令模式以及显示当前所在的行与位置的指示器。

命令行的历史记录与补全

csh、tcsh、ksh、bash……这些shell的用户早在几年前就知道可以召回先前输入的命令，稍做修改再重新下达，这有助于增加生产力。

对于编辑器用户来说更是如此。然而，Unix的vi并没有保存与召回ex命令的功能。

在各种同类品中对此都有解决方案。虽然各有各的保存与召回命令历史记录的方法，但每一种都能用，而且很好用。

除了命令历史记录外，所有的编辑器都可以做某种程度的补全（completion）工作。例如输入某个文件名的开头，接着输入一个特殊字符（如tab），编辑器就会帮你把文件名补全。所有同类品编辑器都能帮我们补全文件名，有些还可以补全其他东西。Vim的此功能细节将于第261页的“关键字与字典词汇补全”中说明。其他编辑器的此项功能细节则将于各自的章节中说明。

任意长度的行与二进制数据

所有同类品都可以处理任意长度的行（注3）。而旧版本的vi常常有一行只能容纳1 000个字符的限制，超过的会被截掉。

所有同类品也都支持8位的数据，这表示它们可以编辑包含任何8位字符的文件。如果需

注3：　最多到C的long类型的最大值2 147 483 647（在32位的计算机上）。

要，甚至可以编辑二进制与可执行文件，这有时候非常有用。你可能需要（也可能不需要）告诉编辑器，文件为二进制的。

nvi

自动处理二进制数据。不需要使用特别的命令行或ex选项。

elvis

在Unix下，不需要对二进制文件做任何特殊处理。在其他系统上，要使用elvis.brf文件来设置binary选项，以避免换行解释的问题（elvis.brf与hex的显示模式，会在第十七章的“有趣的功能”一节中描述）。

Vim

在一行的长度上没有限制。当binary选项未设置时，Vim像nvi一样，会自动处理二进制数据。然而，在编辑二进制文件时，你应该使用-b命令行选项或设置:set binary，如此可设置其他Vim选项，使得编辑二进制文件更容易。

vile

会自动处理二进制数据。不需要使用特别的命令行或ex选项。

最后，还有一个小细节。传统的vi总是在文件最后加上一个换行符。在编辑二进制文件时，这可能会造成问题。nvi与Vim默认情况下与vi兼容，故也会加上换行符。在Vim中，你可以设置binary选项以防止这种情形发生。elvis与vile都不会加上这个换行符。

不限次数的撤销

Unix下的vi只允许你撤销上一次的改变，或者恢复当前这一行的原始状态（在你做任何改变前的状态）。所有的同类品都提供了“不限次数的撤销”功能，可以持续撤销你做的改变，一直到开始任何编辑前的状态。

增量搜索

使用增量搜索（incremental search）时，我们输入搜索模式的同时，编辑器也在文件中移动光标。只有当你最后键入了ENTER，搜索才算结束（注4）。如果你以前从来没使用过这项功能，刚开始可能很不习惯，但是一段时间后就没问题了，习惯后甚至会怀疑以前没有这项功能要怎么工作。

nvi、Vim、elvis均可启用增量搜索的选项， vile则使用两个特殊的vi模式命令。vile可以编译成不支持增量搜索，默认情况下为支持。表8-6显示了每一种编辑器提供的选项。

注4： Emac一直都有增量搜索功能。

表8-6：增量搜索

编辑器	选项	命令	作用
nvi	searchincr		光标在输入同时移动，永远位于搜索到的匹配文本的第一个字符
Vim	incsearch		光标在输入同时移动。Vim会高亮显示匹配当前输入的文本
elvis	incsearch		光标在输入同时移动。elvis会高亮显示匹配当前输入的文本
vile		^X S, ^X R	光标在输入同时移动，永远位于搜索到的匹配文本的第一个字符。^X S往前执行文件的增量搜索，^X R则往回增量搜索。

左右滚动

vi与大多数同类品默认情况下是把过长的行绕排到屏幕的下一行。也就是说，逻辑上的一行可能会占据屏幕上的许多行。

然而，让过长的行在屏幕的右边消失而不是绕排到下一行，有时可能比较好。光标如果位于这样的一行中又一直往右移，最后将“滚动”屏幕。所有的同类品都有这项功能。一般来说，一个数值选项负责控制横向滚动屏幕一次显示的字符数，一个布尔选项控制行是绕排还是在屏幕边缘消失。vile还有命令按键可以将整个屏幕做横向滚动。表8-7显示了在各种编辑器下如何使用水平滚动。

表8-7：横向滚动

编辑器	滚动量	选项	动作
nvi	sidescroll = 16	leftright	默认情况下为关闭左右滚动。打开时，过长的行会在屏幕边缘消失。屏幕从左往右的滚动量为一次16个字符
elvis	sidescroll = 8	wrap	默认情况下为关闭左右滚动。打开时，过长的行会在屏幕边缘消失。屏幕从左往右的滚动量为一次8个字符
Vim	sidescroll = 0	wrap	默认情况下为关闭左右滚动。打开时，过长的行会在屏幕边缘消失。如果将sidescroll设为0，每次滚动会将光标放在屏幕的中央；否则屏幕从左往右滚动一次的字符数量即为指定数值

表8-7：横向滚动（续）

编辑器	滚动量	选项	动作
vile	sideways = 0	linewrap	默认情况下为关闭左右滚动。打开时，过长的行会绕回。因此，默认是让过长的行在屏幕边缘消失。过长的行会在其左右边界以<与>做标记。如果将sideways设为0，每一次会滚动三分之一个屏幕；否则屏幕从左往右滚动一次的字符数量即为指定数值
		horizscroll	默认情况下为打开左右滚动。打开时，在过长的行中移动光标时，会滚动整个屏幕；如果没有打开，只有当前的行会滚动，较适合显示较慢的状况

vile有两个额外的命令：^X ^R与^X ^L。这两个命令分别会将屏幕往右与往左滚动，而光标会停在当前的位置，即光标不能移动到超出屏幕的地方。

可视模式

一般来说，vi中各种操作的对象都是文本单位（如行、单词或字符）或文本段（从光标当前位置到搜索命令所指定的位置）。例如，d/^}的删除范围是到下一个以右括号开头的行为止。elvis与vile都提供了明确选择特定本地文本以运用动作的方式。尤其是可以选择一个矩形块并对内部的文本做动作！关于Vim的此项功能，请参考第172页“可视模式的移动”一节的说明。其他编辑器则参阅第三部分中各编辑器所属的章节。

模式指示器

想必各位都已熟知vi有两种模式：命令模式与插入模式。但通常只看屏幕不能辨别处在哪一种模式中。还有，如果可以知道当前光标在文件中的位置，而不必使用^G或ex的:=命令，会很方便。

针对这些问题，产生了两个选项：*showmode*与*ruler*。所有同类品都具有相同的选项名称与意义，甚至Solaris的vi也有showmode选项。

表8-8列出了每一种编辑器的特殊模式指示功能。

表8-8：位置与模式指示器

编辑器	使用ruler的显示项目	使用showmode的显示项目
nvi	行与列	插入、更改、替换与命令模式的指示器
elvis	行与列	输入与命令模式的指示器
Vim	行与列	插入、替换与可视模式的指示器
vile	行、列与位于文件整体的百分比	插入、替换与覆盖模式的指示器
vi	无	各种模式指示器，包括打开、输入、插入、附加、更改、替换、替换单一字符与替换模式

图形用户界面版本的elvis会依照当前的模式更改光标的形状。

编程辅助

vi是为编程者而开发的编辑器。它有一些功能，让使用传统Unix的编程者——编写C程序与troff文档的人，工作起来更容易（真正的编程者用troff编写真正的文档）。有好几种同类品骄傲地维持这种传统，并加上许多功能，对“专业用户”来说非常有用（注5）。

其中有两项特别值得一提的功能：

编辑－编译的加速

elvis、Vim与vile允许轻易地利用make取得编译器产生的错误，并自动移到包含错误的那一行。接着你便可以修正这些错误，重新执行make，完全不用离开编辑器。

语法高亮显示

elvis、Vim与vile有能力在不同类型的文件中高亮显示或更改不同语法元素的颜色。

编辑－编译的加速

编程通常包括“编译—测试—调试”周期。你会做一些改变，如编译新代码，然后再做测试与调试。在学习一种新语言时，语法错误是常有的事。如果每一次在编译时都要将编辑器暂停与重启（或者挂起与恢复），实在很让人沮丧。

注5：与“所见即所得”（What You See Is What You Get，WYSIWGY）的观念相反，Unix是“所求即所得”（You Asked For it , You Got it）（感谢Scott Lee）。

elvis、Vim与vile都提供了留在编辑器中编译程序的功能。甚至能进一步取得编译器的输出，并自动跳到包含错误的行（注6）。连续使用这个功能，可以节省时间并增加效率。

下例使用的是elvis。假设你刚开始学习C++，因此从这个公认的天字第一号程序开始：

按键顺序	结果

:w hello.C	`#include <iostream>`
	
	`int main()`
	`{`
	`        std::cout << "hello, world!\n ;`
	`        return 0;`
	`}`
	
	在输入代码后，忘了加上结尾的引号，就将程序写入hello.C。

:make hello	`g++     hello.C -o hello` `hello.C:5: error: missing terminating " character` `hello.C: In function 'int main()':` `hello.C:6: error: expected primary-expression before 'return'` `hello.C:6: error: expected `;' before 'return'` `make: *** [hello] Error 1`  接下来输入:make命令以运行make，然后运行C++编译器。g++的输出里说明了每项错误。  `#include <iostream>`  `int main()` `{` ` std::cout << "hello, world\n ;` ` return 0;` `}` `~` `line 5: missing terminating " character 5,8 Command` make的输出消失得很快，而elvis把状态行替换为第一项错误消息，并把光标置于需要修正的行。

你可以修正错误，存储文件，再重新运行:make，最后终能编译出没有错误的程序。

所有这些编辑器都有类似的功能。它们也会察觉文件的改变，并正确地移动到下一个错误所在的行。关于Vim的这项功能细节，请参考第281页的“用Vim编译与检查错误”。其他编辑器的这项细节，请参考第三部分中的各自章节。

注6： 又是一个Emacs的用户在转换到vi时早已习惯的功能。

语法高亮显示

elvis、Vim与vile都提供了某种语法高亮显示。它们也都提供了语法颜色标记（syntax coloring），可依显示模式（像在X11或Linux控制台下）而更改文件中不同部分的颜色。请参考第272页的“语法高亮显示”一节，了解Vim的语法高亮显示功能细节。其他编辑器请参考第三部分中的各相关章节，以取得更多信息。

编辑器功能一览表

大部分的同类品均支持本章提到的大部分或所有功能。表8-9汇总了每一种编辑器支持的功能。当然，这个表并不完备,细节会在各自的章节中提及。

表8-9：功能一览表

功能	nvi	elvis	vim	vile
多窗口编辑	•	•	•	•
图形用户界面		•	•	•
扩展的正则表达式	•	•	•	•
增强的标签		•	•	•
标签栈	•	•	•	•
任意长度的行	•	•	•	•
8位数据	•	•	•	•
不限次数的撤销	•	•	•	•
增量搜索	•	•	•	•
左右滚动	•	•	•	•
模式指示器	•	•	•	•
可视模式		•	•	•
编辑-编译的加速		•	•	•
语法高亮显示		•	•	•
多操作系统支持		•	•	•

还是原创品最好

许多许多年以来，原始vi的源代码因为没有Unix源代码授权而不能取得。虽然教育机构能以相对较低的价格取得授权，但商用授权一直都非常昂贵。因此促成了本书介绍的众多vi同类品的开发。

从2002 年1月起，V7和32V Unix的源代码可通过开放源码的授权而取得（注7）。有了这项开放，几乎可取得所有为BSD Unix开发的源代码，包括ex与vi。

源代码并未根据运行的系统如GNU/Linux而编译，想移植也相当困难（注8）。幸好，已经有人完成移植的工作。如果你想使用原版“真正的”vi，可以下载源代码并自行构建。请参考*http://ex-vi.sourceforge.net/*以取得更多信息。

预告

第三部分的三章会介绍nvi、elvis与vile。每一章的要点大致如下：

1. 谁编写了这个编译器以及为何而编写。
2. 重要的命令行参数。
3. 在线帮助与其他说明文档。
4. 初始化——程序会读入哪些文件与环境变量及读入顺序。
5. 多窗口编辑。
6. 图形用户界面（如果有的话）。
7. 扩展的正则表达式。
8. 改进的便利功能（标签栈、不限次数的撤销等等）。
9. 编程辅助（编辑-编译的加速、语法高亮显示）。
10. 程序特有的有趣功能。
11. 从哪里可以取得源代码以及可以在哪些操作系统上运行。

    所有这些发行包都是由gzip、GNU zip做压缩。如果你没有gzip，可以在*ftp://ftp.gnu.org/gnu/gzip/gzip-1.2.4a.shar*上取得。而在elvis的 FTP 站点中，untar.c程序是一个非常具有可移植性而且简单的程序，可以在非Unix的系统上解开经gzip压缩的tar文件。

因为每一种程序都在继续发展中，我们并不尝试巨细靡遗地介绍其单一的特色，因为现在的介绍很快就会过时。我们会抓住重点，介绍你最可能需要知道而且在发展过程中最不可能改变的特色。如果你需要知道更详细的功能，在本书之外，则应以该编辑器的在线说明文件作为补充。

注7：　关于进一步的信息，请参考Unix Historical Society网站：*http://www.thus.org*。

注8：　我们已经尝试过，才会知道这一点。

第二部分

Vim

第二部分介绍最受欢迎的vi同类品:Vim（vi improved，“改进版 vi”的意思）。该部分包含下列章节：

- 第九章，*Vim（vi Improved）概述*
- 第十章，*Vim对vi的主要改进*
- 第十一章，*Vim的多窗口功能*
- 第十二章，*Vim脚本*
- 第十三章，*图形化Vim（gvim）*
- 第十四章，*程序员专用的Vim强化功能*
- 第十五章，*其他好用的Vim功能*



第九章

Vim（vi Improved）概述

本书的第二部分专门说明Vim——另一个vi。我们先简短介绍 Vim以及它最值得注意的vi相关技术改良，还会提到一点历史源流。本章末会为新用户介绍特殊的Vim模式与教学工具。后续几章内容包括：

- 强化的vi编辑功能
- 多窗口编辑
- Vim脚本
- Vim图形用户界面（GUI）
- 强化的编程相关功能
- 编辑模式
- 其他好用的功能

Vim是指“vi improved”，“改进版vi”的意思。Bram Moolenaar编写了这个工具，也是它的维护者。到今天，Vim或许是使用最广泛的vi同类品，因而有专用的域名（*vim.org*）。最新版本为7.2。

Vim没有标准或规范委员会的限制，可自由地持续增长其功能。它也拥有自己的社群，在开发周期里，通过功能推荐的提名与投票，社群共同决定新增加的功能以及该调整的现有功能。

受到Bram奉献给这个项目的无穷精力与投票系统的启发，Vim的后续发展也很强劲。它随着计算机工业以及文本编辑需求而相应地成长与改变，并维持自身的价值。例如，它有针对特定上下文的语言编辑功能，最初从C开始，而后逐渐包含C++、Java，现在又加入了C#。

Vim包括许多新功能，使得以许多新程序语言编辑代码更为容易。事实上，许多功能在

本书前一版时尚在预计发布阶段，现在都已完全实现了。过去十年来，编程领域里的变化日新月异，Vim也亦步亦趋地随之改变。

现在到处都看得到 Vim，尤其是在Unix及其变化版（例如BSD与GNU/Linux）中，因此许多Vim的用户都觉得Vim 就是vi的同义词。事实上，许多GNU/Linux的发布版本里都默认安装Vim作为/bin/vi二进制版！

Vim提供vi本身没有但对时下的文本编辑器而言不可或缺的功能，例如容易使用、图形化终端支持、彩色、语法高亮显示及格式化，还能进一步自定义个人选项。

概览

作者与简史（注1）

Bram买了一台Amiga计算机后，开始着手于Vim的编写。身为UNIX用户，他一直使用stevie，一个类似vi但实在不够完美的编辑器。幸好，这个编辑器有源代码，所以他开始制作一个更兼容于vi的编辑器并修正错误。过了一段时间，这个程序变得非常好用，Vim 1.14版发布于Fred Fish disk 591上（Amiga专用的免费软件组合包）。

其他人开始使用这个程序，喜欢上这个程序，并开始协助它的开发。首先是提供给UNIX平台的版本，接着是提供给MS-DOS平台和其他系统的版本；随后，Vim成为使用最广泛的vi同类品之一，一路上也增添越来越多的功能：多级撤销、多窗口……有些功能为Vim所独有，但大部分都受益于其他vi同类品。Vim的目标一直都是为用户提供最好的功能。

今天的Vim是功能最全面的类vi编辑器之一，在线帮助也很丰富。

Vim较不出名的功能——从右向左书写，对于希伯来语、波斯语等语言很有用，也勾勒出Vim的灵活性。Vim的另一个设计目标则是成为一个可靠而且可为专业程序员所依赖的编辑器。Vim很少崩溃，真的崩溃时也可以恢复你做的改变。

Vim的开发继续进行，协助新增功能以及把Vim移植到更多平台的人越来越多，各个不同计算机系统上的移植版本的质量也逐渐改善。Microsoft Windows版本具有对话框与文件选择器，为一大群用户打开了难以记忆的vi命令大门。

注1：　本节内容根据Bram Moolenaar（Vim的作者）所提供的资料改写而成，感谢他的协助。

为何选择 Vim？

因为Vim大幅扩展了传统的vi功能，或许改问“为何不选择Vim？”还比较容易。vi引入的功能由其他产品（vile、elvis、nvi）借用，Vim则接下传承的火炬，继续向前奔跑。Vim勇于扩展功能，有时为了让Vim的工作能在恰当的反应时间内完成，还把处理器的性能推到极限。我们不晓得Bram是否对处理器和内存速率的改善很有信心，相信两者能赶上Vim的需求。幸好现在的处理器与内存都能行有余力地应付最困难的 Vim任务。

与 vi 的比较

Vim的接受度比vi更广泛。至少某些版本的Vim几乎可供所有操作系统采用，而vi只适用于UNIX或类似UNIX的系统。

vi是原始版本，几年来只改变了一点点。它是POSIX倡导者，也很切实地履行它的职责。Vim从vi停下的地方开始，它提供所有的vi功能，然后增加了图形界面与其他功能，例如复合选项与脚本编写，这些功能远远超出vi的原始能力。

Vim发布时附有内置的说明文档，以专门的文本文件目录的形式存储。随意浏览一下这个目录（使用标准的UNIX字词计算工具`wc -c *.txt`），将发现有129个文件，包含了122 000 行说明！这是关于Vim功能范围的第一个暗示。Vim通过它的内部“帮助”命令而取用这些文件——另一个vi不曾具备的功能。稍后我们将再仔细了解Vim的帮助系统，并提出一些扩大Vim学习经验的诀窍与秘技。

比较Vim与vi功能的方式之一，就是更仔细地研究说明文档里包含的目录。在这些文件里，Vim以“not in vi”或“not available in vi”标示出vi不具有的选项、命令与函数。大略阅读一下说明文档（使用快速命令`grep -i 'not.. *in vi'`），即找出了超过700个符合的结果。就算这些结果或多或少有些重复，但显然Vim有很多vi没有的功能。

后续章节将提到一些Vim最有趣的功能。从知名Vim扩展到新功能，我们详述了最好与最受欢迎的生产力提高方式。内容涵盖一般认为好用的强化功能，例如语法颜色高亮显示（syntax color highlighting）。我们也会讨论一些较少为人知道但也同样能增加生产力的好用功能，例如自定义Vim状态行，从而于每次移动光标时即时更新日期与时间。

功能分类

Vim的功能从活动命令横跨到几乎任何文本编辑任务。有些功能只是扩充了用户对vi的希望，有些则是全新功能，未曾出现于vi。如果你还需要其他连Vim也没有的功能，可

以用Vim的内置脚本语言，以无限制地扩充与自定义需求。Vim的部分功能分类如下：

语法扩展（syntax extension）

Vim 能让我们控制缩排以及根据语法为代码文本上色，还有很多选项能定义自动格式。如果你不喜欢高亮显示的颜色，可以自己改变。如果需要特殊形式的缩排，Vim已提供了一些选择；若你的需求更为特别，也可以自定义环境。

程序员辅助（programmer assistance）

虽然Vim并未试着提供所有编程的需求，但已经提供许多一般只会在IDE（Integrated Development Environment，集成开发环境）里出现的功能。从快速的“编辑—编译—调试”周期，到自动补全关键字，Vim的特殊功能不只能让我们快速地编辑——还更有助于编程。

图形用户界面功能（GUI feature）

加上使用鼠标点击的编辑功能后（与现在多数易用的编辑器一样），Vim也为一般大众拓展了可用性。较轻松、较简单的编辑任务若需使用较艰深的功能（所有高级用户的功能），都已加上容易通过 GUI 取用的途径。

脚本编写与插件程序（scripting and plug-in）

你可以编写自己的Vim扩展组件，或从网络下载插件程序；甚至还可以发布你的扩展组件，让其他人使用，为Vim社群贡献一己之力。

初始化（initialization）

Vim使用配置文件来在启动时定义会话（session），这点与vi一样，但Vim扩展了很多可定义行为。我们可以只简单地设置几个选项，像在vi中一样；也可以根据任何你定义的上下文而设计整套个人化选项。举例来说，你可以自己编写初始化文件，以根据所编辑文件所在的目录来预编译代码；或可于启动时从某些实时信息来源中捕获信息并与文本内容整合。

会话上下文（session context）

Vim把会话的信息保存在.viminfo文件中。你在重新查看与编辑文件的时候，可曾想过“当前工作到哪里了？”这项功能解决了你的疑问！它可以定义在各会话中要保留多少信息与何种信息。例如，可以定义“最近文档”或最新编辑文件的数量以利跟踪，每个文件可记忆的编辑动作数量（删除、改变等等），命令历史记录可记忆的命令数量，以及由前次编辑动作（放置、删除等等）要保存的缓冲区与文本行数量。Vim 不只记忆最后一次文件会话的编辑行为，也在切换文件时记忆基本信息。如此对于“捕获A文件的数行文本（使用y〔拖动〕或d〔删除〕），而后放置到B文件”的编辑动作甚是便利。从一个文件切换到另一个文件时，在未命名缓冲区中的任何内容仍会由Vim记忆并可取用。还有，Vim也会记忆最后一个搜索模

式，所以我们开始新会话找寻上一次使用的搜索模式时，只需利用n命令（寻找下一个）。

Vim也会记忆我们在每个最近编辑文件中的行位置。如果在光标位于第25行时退出编辑会话，则下次编辑时Vim会把你带回第25行。

后处理（postprocessing）

除了定义在会话前执行的函数，Vim也能让我们定义编辑文件后的行为。你可以编写清除例程以删除编译时累积的临时文件，或者在文件写回硬盘前实时编辑。我们对任何编辑后行为拥有完全的控制权。

状态转换（transition）

Vim也管理状态转换。当我们在会话中由一个缓冲区移动到另一个缓冲区或从一个窗口移动到另一个窗口（两者通常是一回事）时，Vim也会自动地做一些事前事后的整理工作。

透明的编辑（transparent editing）

Vim会检测并自动解开压缩文档。例如，可以直接编辑如`myfile.tar.gz`这样的压缩文档。你甚至可编辑目录。Vim让你可使用熟悉的导航（navigation）命令于目录中导航并选择欲编辑的文件。

元信息（meta-information）

Vim提供4种很方便的只读寄存器，让用户可捕获用于“放置”的元信息，包括当前文件名（`%`）、替换文件名（`#`）、前次于命令行执行的命令（`:`）、前次插入的文本（`.`）。

黑洞寄存器（black-hole register）

这是一项不知名但很有用的编辑寄存器扩展功能。一般而言，删除文本即为使用rotation scheme把该文本放入缓冲区。rotation scheme在循环利用原有删除内容以取回删除的内容时很有用。Vim提供了“黑洞”寄存器，用于丢弃删除内容而不影响正常寄存器中已删除文本的 rotation。如果你是Unix用户，它就是Vim版本的`/dev/null`。

关键字补全（keyword completion）

Vim 采取上下文相关的补全（context-sensitive completion）规则帮我们补全已输入部分内容的词汇。例如，Vim会于字典或包含某语言专用关键字的文件中寻找词汇。

Vim能通过设置`compatible`选项（`:set compatible`）回到与vi兼容的模式。大多数时候，我们应该都想利用Vim的额外功能，但它还是贴心地提供了向下兼容的选项。

理念

Vim的理念与vi相近。两者均为编辑工作带来强大威力与优雅；两者均依赖modality（命令模式与输入模式）；而且都把编辑工作交给键盘：也就是说，用户的编辑工作可以快速而有效率地进行，且完全不需用到鼠标（或^X ^C）。我们习惯称此为“接触式编辑”（touch editing），类似于“接触式键入／盲打”（touch typing），这反映出它们各自增进了与任务相关的速度与效率。

Vim更扩充了这份理念，它为经验不足的用户提供功能（GUI、可视高亮显示模式），也为高级用户提供强化选项（脚本编写、扩展的正则表达式、可配置的语法以及可配置的缩排方式）。

至于喜欢编码的超级用户，Vim已附上源代码。用户可自由（并受鼓励地）增进已改善的各项功能。就理念而言，Vim达成了所有用户需求的平衡。

取得Vim

如果你的操作环境是Unix的变体（包括Mac OS X），恭喜你，Vim可能已经安装好了。如果它可通过预定义的PATH环境变量取得与执行，则于shell命令行输入vim后即可打开Vim窗口。如果出现如下典型的 UNIX 错误消息：

```
sh: command not found: vim
```

则改为键入vi，看看Vim的欢迎消息是否出现。你的安装版本可能以Vim替换了vi。

许多系统附有老版Vim。因此，即使你已经有Vim可用，本节在你安装最新版时还是颇有用处。进入编辑器后，先以:version命令确认正在运行Vim，同时确认版本。Vim将呈现如下画面：

```
:version
VIM - Vi IMproved 7.0 (2006 May 7, compiled Aug 30 2006 21:54:03)
Included patches: 1-76
Compiled by corinna@cathi
Huge version without GUI. Features included (+) or not (-):
+arabic +autocmd -balloon_eval -browse ++builtin_terms +byte_offset +cindent
-clientserver -clipboard +cmdline_compl +cmdline_hist +cmdline_info +comments
+cryptv +cscope +cursorshape

 ...

 +profile -python +quickfix +reltime +rightleft -ruby +scrollbind +signs
+smartindent -sniff +statusline -sun_workshop +syntax +tag_binary +tag_old_static
-tag_any_white -tcl +terminfo +termresponse +textobjects +title -toolbar
+user_commands +vertsplit +virtualedit +visual +visualextra +viminfo +vreplace
+wildignore +wildmenu +windows +writebackup -X11 -xfontset -xim -xsmp
```

```
-xterm_clipboard -xterm_save

 system vimrc file: "$VIM/vimrc"
    user vimrc file: "$HOME/.vimrc"
     user exrc file: "$HOME/.exrc"
  fall-back for $VIM: "/usr/share/vim"
Compilation: gcc -c -I. Iproto -DHAVE_CONFIG_H     -g -O2
 Linking: gcc -L/usr/local/lib -o vim.exe       -lncurses  -liconv -lint
```

以上输出内容将于第十章中协助各位以自定义选项编译Vim的上下文中讨论。

注意： 有趣的是，某位作者的安装了OS X 10.4.10版的Mac Mini中，不只vi命令会调用Vim，说明文档（manpage）也把 Vim写入文档里！

如果你在自己的系统上还是找不到Vim，可以先尝试寻找下列常见目录，再考虑下载与安装它。如果你找到了可执行文件，请把它的目录加入你的PATH，然后就万事俱备了：

/usr/bin（应该已在 PATH 里）
/bin（应该也在 PATH 里）
/opt/local/bin
/usr/local/bin

如果连前述目录下都没有看似Vim的文件，你的系统大概就真的没有安装 Vim。幸好，Vim为许多平台提供了多种取用模式，而且（通常相对地）算是较容易取得与安装。接下来的小节将引导各位取得适用于各自所用平台的Vim。我们将依序讨论下列平台的Vim安装方式：

- Unix 及其变体，包括GNU/Linux
- Windows XP、2000、Vista
- Macintosh

取得Unix与GNU/Linux环境中的Vim

许多现在的Unix环境中已经附有某一版的Vim。大多数GNU/Linux发行包均把默认的vi位置/bin/vi链接到Vim可执行文件。大部分Unix用户不需另行安装Vim。

因为有太多的Unix变体，有些变体本身又有许多变化（例如Sun Solaris HP-UX、*BSD、所有GNU/Linux的发行版本），最直接也最推荐的Vim取得方式就是下载它的源文件，然后编译并安装。

注意：本处描述的安装过程需要一个能够编译源代码的开发环境。虽然大多数Unix变体已提供编译器与相关工具，有些（特别是最近的GNU/Linux发行版本Ubuntu）却需要下载并安装附加包，才能体验编译代码的乐趣。

Vim网站上推荐一个最新的安装程序aap，也提供它的链接与概述。因为aap是新出来的工具，而原有的下载与编译安装方式能良好运作，故我们不采用aap作为安装程序。不过等各位看到本书时，aap的使用或许已经是常识了。

另外还有预先打包好的Vim bundle，为GNU/Linux（Red Hat RPM、Debian pkgs）、IRIX（SoftwareManager）、Sun Solaris（Companion Software）、HP-UX提供简易标准安装。Vim的主页提供有上述系统的信息链接。

Vim的源代码可从Vim网站（*http://www.vim.org*）上取得。源代码用tarball打包压缩，存储为GZIP（.gz）或BZIP2（.bz2）格式。实际上，时下所有操作系统都能辨认并处理GZIP文件，而大多数Unix变体都有处理BZIP2的工具。下载源代码并依照下列顺序解开压缩文件，如果你安装的是不同版本则替换下载的文件的名称：

.gz*文件*

```
$ gunzip vim-7.1.tar.gz
```

.bz2*文件*

```
$ bunzip2 vim-7.1.tar.gz
```

命令完成后，vim-7.1.tar文件（或是任何表示下载版本的类似文件）仍然存在。现在解压缩tar 文件：

```
$ tar xvf vim-7.1.tar
vim71
vim71/README.txt
vim71/runtime
vim71/README_UNIX.txt
vim71/README_lang.txt
vim71/src
vim71/Makefile
vim71/Filelist
vim71/README_src.txt
 ...
vim71/runtime/doc/vimtutor-ru.1
vim71/runtime/doc/xxd-ru.1
vim71/runtime/doc/evim-ru.UTF-8.1
vim71/runtime/doc/vim-ru.UTF-8.1
vim71/runtime/doc/vimdiff-ru.UTF-8.1
vim71/runtime/doc/vimtutor-ru.UTF-8.1
vim71/runtime/doc/xxd-ru.UTF-8.1
```

现在可以删除vim-7.1.tar了。再切换到以tar命令创建的Vim目录下：

```
$ cd vim71
```

configure文件是个脚本，用于配置编译参数。大多数配置工作应该留给脚本，由它检查主机环境并根据系统安装的软件而决定功能的打开与否。

此时可以决定是否使用默认值或选择性地选用功能。例如在编译时选择打开perl接口（否则configure脚本不能用其他方式实现），以备未来安装perl脚本编写工具之需：

```
$ ./configure --enable-perlinterp
```

或者决定不使用perl接口并用configure选项关闭这项功能：

```
$ ./configure --disable-perlinterp
```

警告： 当前的Vim版本提供稍稍不一样的自定义安装。不再是把所有的--disable-XXX与--enable XXX放到configure选项，INSTALL指示我们应在feature.h文件中修正选项。我们比较推荐在Vim的配置文件里编译可取得的选项（参考README.txt）与自定义编译需求，除非你有修改这个文件的编译理由。

一般configure的输出结果（采用默认值，没有任何选项）如下所示：

```
$ configure
/home/ehannah/Desktop/vim/vim71/src
configure: loading cache auto/config.cache
checking whether make sets $(MAKE)... (cached) yes
checking for gcc... (cached) gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
...
checking for NLS... no "po/Makefile" - disabled
checking for dlfcn.h... (cached) yes
checking for dlopen()... no
checking for dlopen() in -ldl... yes
checking for dlsym()... yes
checking for setjmp.h... (cached) yes
checking for GCC 3 or later... yes
configure: creating auto/config.status
config.status: creating auto/config.mk
config.status: creating auto/config.h
config.status: auto/config.h is unchanged
```

现在以make实用程序构建Vim：

```
$ make
Starting make in the src directory.
If there are problems, cd to the src directory and run make there
cd src && /usr/local/lib/cw/make first
```

```
/home/ehannah/Desktop/vim/vim71/src
make[1]: Entering directory `/home/ehannah/Desktop/vim/vim71/src'

gcc -c -I. -Iproto -DHAVE_CONFIG_H -g -O2 -o objects/
    charset.o charset.c
gcc -c -I. -Iproto -DHAVE_CONFIG_H -g -O2 -o objects/
    diff.o diff.c
gcc -c -I. -Iproto -DHAVE_CONFIG_H -g -O2 -o objects/
    digraph.o digraph.c
gcc -c -I. -Iproto -DHAVE_CONFIG_H -g -O2 -o objects/
    edit.o edit.c

 ...

make[2]: Entering directory `/home/ehannah/Desktop/vim/vim71/src'
creating auto/pathdef.c
gcc -c -I. -Iproto -DHAVE_CONFIG_H -g -O2 -o objects/
    pathdef.o auto/
    pathdef.c
make[2]: Leaving directory `/home/ehannah/Desktop/vim/vim71/src'
link.sh: Using auto/link.sed file to remove a few libraries
  gcc -o vim objects/buffer.o objects/charset.o objects/diff.o
  objects/digraph.o objects/edit.o objects/eval.o objects/ex_cmds.o
  objects/ex_cmds2.o objects/ex_docmd.o objects/ex_eval.o
  objects/ex_getln.o objects/fileio.o objects/fold.o objects/getchar.o
  objects/hardcopy.o objects/hashtab.o objects/if_cscope.o
  objects/if_xcmdsrv.o objects/main.o objects/mark.o objects/memfile.o
  objects/memline.o objects/menu.o objects/message.o objects/misc1.o
  objects/misc2.o objects/move.o objects/mbyte.o objects/normal.o
  objects/ops.o objects/option.o objects/os_UNIX.o objects/pathdef.o
  objects/popupmnu.o objects/quickfix.o objects/regexp.o objects/screen.o
  objects/search.o objects/spell.o objects/syntax.o objects/tag.o
  objects/term.o objects/ui.o objects/undo.o objects/window.o
  objects/netbeans.o objects/version.o -lncurses -lgpm -ldl

link.sh: Linked fine with a few libraries removed
cd xxd; CC="gcc" CFLAGS=" -g -O2" \
    /usr/local/lib/cw/make -f Makefile
/home/ehannah/Desktop/vim/vim71/src/xxd
make[2]: Entering directory `/home/ehannah/Desktop/vim/vim71/src/xxd'
gcc -g -O2 -DUNIX -o xxd xxd.c
make[2]: Leaving directory `/home/ehannah/Desktop/vim/vim71/src/xxd'
make[1]: Leaving directory `/home/ehannah/Desktop/vim/vim71/src'
```

如果一切进行顺利，在你的src目录下应该有可执行的Vim了。虽然Vim已可使用，但我们还需指定完全路径名称或把Vim的目录加入每个用户的可执行路径（executable path）来调用它。如果你不能以系统管理员的身份安装程序，则需执行此步骤。

想把Vim安装为本机所有用户的公共资源，你必须拥有系统管理员（root）的权限。如果可以，请切换至root身份并输入：

```
# make install
Starting make in the src directory.
If there are problems, cd to the src directory and run make there
cd src && make install
/home/ehannah/Desktop/vim/vim71/src
make[1]: Entering directory `/home/ehannah/Desktop/vim/vim71/src'
if test -f /usr/local/bin/vim; then \
  mv -f /usr/local/bin/vim /usr/local/bin/vim.rm; \
  rm -f /usr/local/bin/vim.rm; \
fi
cp vim /usr/local/bin
strip /usr/local/bin/vim
chmod 755 /usr/local/bin/vim
cp vimtutor /usr/local/bin/vimtutor
chmod 755 /usr/local/bin/vimtutor
/bin/sh ./installman.sh install /usr/local/man/man1 "" /usr/local/
    share/vim /usr/local/share/vim/vim71 /usr/local/share/vim ../
    runtime/doc 644 vim vimdiff evim
installing /usr/local/man/man1/vim.1
installing /usr/local/man/man1/vimtutor.1
installing /usr/local/man/man1/vimdiff.1

 ...

if test -d /usr/local/share/icons/hicolor/48x48/apps -a -w /usr/
    local/share/icons/hicolor/48x48/apps \
    -a ! -f /usr/local/share/icons/hicolor/48x48/apps/gvim.png; then \
   cp ../runtime/vim48x48.png /usr/local/share/icons/hicolor/48x48/
     apps/gvim.png; \
fi
if test -d /usr/local/share/icons/locolor/32x32/apps -a -w /usr/
    local/share/icons/locolor/32x32/apps \
    -a ! -f /usr/local/share/icons/locolor/32x32/apps/gvim.png; then \
   cp ../runtime/vim32x32.png /usr/local/share/icons/locolor/32x32/
     apps/gvim.png; \ x
fi
if test -d /usr/local/share/icons/locolor/16x16/apps -a -w /usr/
    local/share/icons/locolor/16x16/apps \
    -a ! -f /usr/local/share/icons/locolor/16x16/apps/gvim.png; then \
   cp ../runtime/vim16x16.png /usr/local/share/icons/locolor/16x16/
       apps/gvim.png; \
fi
make[1]: Leaving directory `/home/ehannah/Desktop/vim/vim71/src'
```

安装到此完成，只要其他用户的PATH变量设置正确，他们应该就都能使用Vim。

取得Windows环境中的Vim

对于Microsoft Windows，主要有两种安装Vim的选择。第一种是自行安装可执行文件——gvim.exe，其可于Vim的网站上取得。下载并运行此文件，它会自动完成后续工作。我们曾在不同的Windows机器上安装这个程序，它都能准确地工作。二进制文件应能正确地安装在 Windows XP、2000、NT、ME、98、95 上。

注意： 在安装过程中，会有DOS窗口弹出以警告某项东西未被验证，但我们没看到出过什么问题。

Windows用户的第二种选择则是安装Cygwin（*http://www.cygwin.com/*），一个把常用GNU 工具移植到Windows平台的套件。Cygwin是用于Unix平台上的几乎所有主流软件的完整实现。Vim是Cygwin安装时的标准配备，可从Cygwin shell窗口运行。

以Cygwin使用Vim

基于文本的控制台Vim能在Cygwin上运作良好，但Cygwin的gvim则需X Window System server才能运行，如果其启动时没有这个服务器，将大幅降低（基于文本的）Vim的性能。

为了让Cygwin的gvim运作（假设想在本地屏幕上运行），请在Cygwin的shell里通过命令行启动X server，如下所示：

```
$ X -multiwindow &
```

选项-multiwindow告诉X server，让Windows管理Cygwin应用程序。还有许多其他使用Cygwin的X server的方式，但已经超出本书范围了。安装Cygwin的X server也已超出本书范围，如果它尚未被安装，请参考Cygwin网站以了解更多信息。应该会有一个"X"图标出现在Windows的系统列表中，它是X server正在运行的证明。

同时安装Cygwin的Vim与*www.vim.org*的Vim只会造成困扰。需参考Vim配置的配置文件可能被放在不同地方，因此造成好像有两个相同版本的Vim，却以完全不同选项启动的感觉。例如，Cygwin和Windows可能就对Vim的主目录持不同意见。

取得Macintosh环境中的Vim

Mac OS X出售时就已安装了Vim 6.2版，但没有其他GUI版本。用户可自行下载.tar.bz2 文件，以编译具有GUI的6.4版或7.1版。

不过，下载源文件时，它的维护人员推荐从CVS（源码控制系统）下载，以确保取得最新的源代码以及最新的补丁程序。这做起来并不困难，但通过命令行下载，对于新接触的用户而言可能是个非常新颖的观念。

下载文件后，安装过程与Unix的编译与安装过程非常相似，请参考第155页的"取得Unix与GNU/Linux环境中的Vim"。

其他操作系统

Vim的网站上列出了其他显然可让Vim运行的环境，但也警告用户要自己承担风险。Vim可供如下系统使用：

- QNX，一种实时操作系统（Real-Time Operating System，RTOS）
- Agenda
- Sharp Zaurus，一种基于Linux的手提设备系统
- HP Jornada，一种基于Linux的手提设备系统
- Windows CE，一种Windows版的手提设备系统
- Alpha 上的Compaq Tru64 Unix
- Open VMS，Digital的VMS附加POSIX
- Amiga
- OS/2
- RISC OS，一种基于精简指令集计算机（Reduced Instruction Set Computer，RISC）的 OS
- MorphOS，一种建立在Quark内核之上的基于Amiga OS的OS

给新用户的帮助工具与简易模式

请记住，vi与Vim都需要用户花一点时间学习，Vim提供了一些令使用简便的功能：

Graphical Vim（gvim）

: 当用户调用gvim命令时，将呈现一个丰富的图形化窗口，为Vim提供某些时下图形用户界面（GUI）程序上受欢迎的鼠标点击(point-and-click)功能。在很多环境中，gvim是个不同的二进制文件，由编译Vim创建并已打开所有的GUI选项。亦可使用vim -g调用它。

"Easy"Vim（evim）

: evim命令取代了某些标准vi功能的简单行为。若是不熟悉vi的用户，可能会觉得这是较直接的文件编辑方式；而专家级的用户大概不会觉得这种模式有何简单，因为他们已经习惯了标准的vi行为。亦可使用vim -e调用它。

vimtutor

: Vim已附有vimtutor，一个启动Vim并附上特殊帮助文件的命令。这种调用Vim的方式为用户提供了学习这个编辑器的起点。完成vimtutor大概需要30分钟。

小结

vi仍是Unix上的标准文本编辑工具。它在当年几乎是完全创新的事物，具有双重模式与触碰式编辑原理。Vim延续vi停下的地方，它是强大编辑与文本管理的下一个创新脚步：

- Vim扩展了vi，从旧有编辑器的绝佳标准集上逐步构建。虽然也有其他根据原始vi构建的编辑器，但Vim已从中脱颖而出，成为最受欢迎、使用最广泛的vi同类品。
- Vim的功能比vi多得多，足以成为新的标准。
- Vim为初学者设计，也为高级用户设计。它为初学者提供许多学习工具及"easy"模式；为专家则提供了强大的vi扩展功能；还有用于增强并调整Vim以符合自身需要的平台。
- 到处都有Vim在运行。如本章的讨论内容，在不能取得Vim的环境中，自然有人站出来，把它移植到最好用的OS平台上。Vim或许不是真的到处都有，但也差不多了！
- Vim是自由软件。还有，如本书前一版所言，Vim是个慈善软件（charityware）。Bram Mollenaar对于创建、改进、维护以及延续Vim所作的努力，在自由软件市场上真是非常值得表彰的功绩。如果你喜欢他的作品，Bram也希望你了解他最大的理想：帮助乌干达的儿童。请访问*http://iccf-holland.org/*以了解更多信息，或使用Vim 内置的帮助命令，参考"uganda"标题（**:help uganda**）。



第十章

Vim对vi的主要改进

Vim对vi做了大量改进，从多彩的语法定义，到成熟的脚本编写等等，不一而足。如果说vi是表现优秀的（也确实优秀），Vim则是好得不可思议。本章将讨论Vim如何添加用户一直抱怨但vi一直没有的功能。先列举如下：

- 内置帮助功能
- 启动与初始化选项
- 新的移动命令
- 扩展的正则运算式
- 扩展的撤销
- 自定义可执行文件（executable）

内置帮助功能

前一章提过，Vim的说明文档有超过10万行，几乎所有说明都可通过Vim内置的帮助工具取得。使用它的最简单的形式就是调用:help命令（它的有趣之处在于让用户首次接触到Vim的多窗口编辑）。

虽然:help命令很方便，但却不甚实用，因为内置的帮助功能需要稍微了解vi的导航（navigation）技巧：想真正有效率地使用，用户必须知道如何在标签间前进与后退。下面将列出帮助画面的导航概览。

:help将出现类似下面的画面：

```
*help.txt*    For Vim version 7.0.    Last change: 2006 May 07

                    VIM - main help file
                                                  k
```

```
      Move around: Use the cursor keys, or "h" to go left,          h      l
                   "j" to go down, "k" to go up, "l" to go right.       j
Close this window: Use ":q[Enter]".
 Get out of Vim: Use ":qa![Enter]" (careful, all changes are lost!).

Jump to a subject: Position the cursor on a tag (e.g. |bars|) and hit CTRL-].
   With the mouse: ":set mouse=a" to enable the mouse (in xterm or GUI).
                   Double-click the left mouse button on a tag, e.g. |bars|.
        Jump back: Type CTRL-T or CTRL-O (repeat to go further back).

Get specific help: It is possible to go directly to whatever you want help
                   on, by giving an argument to the |:help| command.
                   It is possible to further specify the context:
                                                  *help-context*
                         WHAT                  PREPEND    EXAMPLE ~
                     Normal mode command      (nothing)   :help x
```

幸好，Vim考虑到初学者可能有潜在的导航问题，而贴心地让画面一开始就有基础导航方式指示，甚至还教用户如何退出帮助画面。我们推荐以这个帮助说明为起点，而且鼓励大家多花点时间探索帮助画面。

熟悉帮助画面后，可于Vim的命令行里利用tab自动补全的功能扩大探索范围。对于任何以位于命令提示符（:）后的命令，只要按下Tab键，就会依据当时的上下文，在命令行里予以补全。例如：

```
:e /etc/termc[TAB]
```

在任何Unix系统里将展开为：

```
:e /etc/termcap
```

:e命令暗示此命令的参数为一个文件，所以命令自动补全机制会寻找匹配部分文件名的文件以完成输入。

但:help有自己的上下文，也包含了帮助主题。我们输入的部分主题字符串可能会与出现在Vim帮助主题里的某个字符串符合。我们推荐大家学习并使用这项功能，除了节省时间，还可能会发现不曾听过的有趣功能。

假设你想了解如何分割屏幕。首先键入：

```
:help split
```

然后按下Tab键。在当前的版本中，help命令将在下列字符串间循环：split(); :split; :split_f; splitview; splitfind; 'splitright'; 'splitbelow'; g:netrw_browse_split; :dsplit; :vsplit; :isplit; :diffsplit; +vertsplit;……不胜枚举。想看任何一项主题的帮助时，可在该主题被高亮显示时按下ENTER键。

你不只可看到想寻找的主题（:split），也会发现一些原本没想到的东西，例如：vsplit——“垂直分割”命令。

启动与初始化选项

Vim使用不同机制于启动时设置环境。它会检查命令行选项，也会自我检查（如何被调用、以何名称被调用等等）。另有其他编译好的二进制文件以供不同需要使用（例如GUI或文本窗口）。Vim也使用一系列初始化文件，用于定义并调整难以数计的行为组合。这些选项实在多到难以全面讲解，我们只讨论最有趣的几项。在接下来的小节中，将介绍 Vim 的启动顺序以及下列主题：

- 命令行选项
- 与命令名称相关的行为
- 配置文件（分为系统整体与个别用户的）
- 环境变量

这一节先介绍启动Vim的一些方式。想知道更多选项的众多细节，请善用帮助命令：

```
:help startup
```

命令行选项

Vim的命令行选项提供了灵活性与威力。有些选项可调用额外功能，有些则可覆盖并抑制默认行为。我们在讨论命令行语法时，将视其用于典型的Unix环境中。单一字母选项前需加上-（一个连字符），如-b，表示允许编辑二进制文件。类似一般词汇的选项前则加上--（两个连字符），如--noplugin，表示覆盖载入插件的默认行为。只包含两个连字符的命令行参数，则表示该命令行并未包含选项（此为标准Unix行为）。

在命令行选项后，我们可以选择列出一或多个待编辑文件的文件名（事实上，文件名可为“-”，这将造成有趣的状况，让Vim知道输入来自标准输入，*stdin*。这在稍后将有讨论，但鼓励大家自行研究它的使用方式）。

接下来列出一部分Vim有但vi没有的命令行选项（Vim已包含所有vi选项）：

-b

在二进制模式中编辑。它的功能从名称上已可得知，而且很棒。编辑二进制文件是种需要培养的嗜好，但在编辑多数工具接触不到的文件时，它是种很强大的方式。用户最好阅读Vim关于二进制文件的帮助说明。

`-c` *command*

*command*将被视为ex命令执行。vi也有相同的选项，但Vim在一条命令中最多允许加入10个-c选项。

`-C`

以兼容（vi）模式运行Vim。因某些很明显的原因，本选项不可能出现在vi中。

`-cmd` *command*

*command*在vimrc文件前执行。这是-c选项的长形式。

`-d`

以diff模式启动。Vim执行对2~4个文件的diff功能以及可设置选项来简化文件差异的查看（例如scrollbind、foldcolumn等）。

Vim使用操作系统可用的diff命令，即Unix系统上的diff。Windows版则可下载另行提供的可执行文件，然后Vim才能执行diff任务。

`-E`

以改进的ex模式启动。例如可将改进的ex模式用于扩展正则运算式。

`-F`或`-A`

分别表示波斯语（Farsi）或阿拉伯语（Arabic）模式。使用此两种模式必须有键盘与字符映射的帮助，并将由右至右绘制屏幕。

`-g`

启动gvim（GUI）。

`-m`

关闭写入选项。此时将不能修改缓冲区。

`-o`

所有文件均各自打开窗口。可指定欲打开的窗口数量（以整数指定）。命令行中列出的文件分别于指定数量的窗口中打开（若有剩下的文件，则放入Vim的缓冲区）。如果指定的窗口数量超出文件数量，Vim将打开空白窗口，以满足数量要求。

`-O`

与-o一样，但打开垂直分割的窗口。

`-y`

以easy模式运行Vim，可为初学者把选项设置为较直观的行为。“easy”或许能帮助不熟悉Vim的人，但经验丰富的用户只会被这个模式弄得一头雾水。

-Z

　　以restricted模式运行。基本上关闭所有外部接口并防止对系统功能的访问。例如，用户不可使用!G!sort从缓冲区的当前文本行中寻找到文件的结尾，也将不能使用过滤器sort。

接下来使用服务器Vim的远程实例示范一系列相关选项。remote命令告诉远程的Vim（不见得在同一台主机上执行）编辑一份文件，或在远程服务器上计算一个运算式。服务器相关命令告诉Vim该把内容传向哪个服务器，亦可自我声明为服务器。serverlist则列出可用的服务器：

-remote *file*
-remote-silent *file*
-remote-wait *file*
-remote-send *file*
-servername *name*
-remote-expr *expr*
-remote-wait-silent *file*
-remote-tab
-remote-send *keys*
-remote-wait-silent *file*
-serverlist

若想了解所有命令行选项的更复杂内容，包括完整的vi设置，请参考第381页的“命令行语法”。

与命令名称相关的行为

Vim有两种版本，图形版（在Unix变体中使用X Window System，在其他操作系统中则用系统本身的 GUI）与文字版，两种都可设置启动时的字符子集。Unix用户只需简单地使用下列命令之一，即可获得所需行为：

vim

　　启动文字版Vim。

gvim

　　启动图形版Vim。在许多环境中，gvim是个不同的Vim二进制文件，已于编译时打开所有GUI选项，就像vim -g（在 Unix 环境中，gvim需要 X Window System）。

view, gview

　　以只读模式启动Vim或gvim。与vim -R相同。

rvim

以restrictive模式启动Vim。所有对shell命令的外部访问均被禁止，以^Z命令中止编辑会话的能力也被禁止。

rgvim

与rvim相同，但用于图形版。

rview

与view类似，但于restricted模式中启动。在restricted模式中时，用户不能访问过滤器（filter）、外部环境（outside environment）或OS功能。与vim -Z相同（-R选项只会调用先前描述的只读效果）。

rgview

与rview相同，但用于图形版。

evim, eview

于编辑或只读查看时使用“easy”模式。Vim设置了一些选项与功能让不熟悉 Vim规范的人觉得使用上较为直观。与vim -y相同。有经验的用户大概不会觉得这个模式“简单”，因为其已经熟悉标准vi行为。

请注意，这个命令没有类似的*gXXX*版，因为gvim显然已被视为简单的操作方式，至少学习时有可预测的点击行为。

vimdiff, gvimdiff

以“diff”模式启动并比较输入文件的差异。稍后于第295页的“有何差异？”一节中将有更深入的说明。

ex, gex

使用老式的ex行编辑模式。在处理脚本时很有用。与vim -e相同。

Windows 用户可于程序列表（“开始”菜单）中访问类似的 Vim 版本。

系统与用户的配置文件

Vim依照特定顺序寻找初始化的信号。它执行找到的第一组指令（可能以环境变量或文件的形式出现），然后开始编辑工作。所以，Vim在下列清单中遇到的第一个项目，就是清单中被执行的唯一项目。顺序如下：

1. VIMINIT。它是环境变量，如果不为空，Vim即把它的内容视为ex命令而执行。
2. 用户的vimrc文件。vimrc（Vim资源）初始化文件是个跨平台的文件，但因为操作系统与平台间的微妙差异，Vim将以下列顺序在不同位置寻找此文件：

$HOME/.vimrc (Unix, OS/2, and Mac OS X)

s:.vimrc (Amiga)

home:.vimrc (Amiga)

$VIM/.vimrc (OS/2 and Amiga)

$HOME/_vimrc (DOS and Windows)

$VIM/_vimrc (DOS and Windows)

3. exrc选项。如果设置了Vim的exrc选项，它会寻找三个额外的配置文件：[._]vimrc、[._]vimrc、[._]exrc。

vimrc文件是配置Vim的编辑特性较好的地方。差不多任何Vim选项都能在此文件中被设置为打开或关闭，而且它特别适合设置全局变量与定义函数、缩写、按键映射……以下是vimrc的一些须知事项：

- 注释以双引号（"）开始，可位于一行的任何位置。所有位于双引号后的文本，包括双引号，都会被忽略。
- 可用（亦可不用）冒号表示ex命令。例如，set autoindent其实与:set autoindent完全一样。
- 如果一大组选项定义能分开为不同行，这个文件将较容易管理。例如：

  ```
  set terse sw=1 ai ic wm=15 sm nows ruler wc=<Tab> more
  ```

  与下面相同：

  ```
  set terse " short error and info messages
  set shiftwidth=1
  set autoindent
  set ignorecase
  set wrapmargin=15
  set nowrapscan " don't scan past end or top of file in searches
  set ruler
  set wildchar=<TAB>
  set more
  ```

  请看第二组命令是否容易阅读多了。第二种方式也较容易维护，有过许多次删除、插入、于配置文件中调试设置时暂时以注释停止某行功能的经验时就能体会。例如，我们可能想暂时于启动配置中禁用行编号的功能，此时只要简单地在配置文件中的set number行前加上双引号即可。

环境变量

有许多环境变量影响到Vim的启动行为，甚至影响某些编辑会话的行为。这里列出最为明显的且在未配置时会以默认值处理的环境变量。

如何设置环境变量

登录时（在Unix中调用*shell*）可设置变量以反映或控制命令环境的其行为。环境变量的威力特别强大，因为它们会影响在命令环境里调用的程序。接下来的介绍并非Vim专用，它们可用于设置在命令环境中你想设置的任何环境变量。

Windows 系统

设置环境变量时：

1. 调出控制台。
2. 双击“系统”图标。
3. 点击“高级”选项卡。
4. 点击“环境变量”按钮。

以上操作的结果将跳出一个窗口，窗口内分隔成两个环境变量区域——用户与系统。新手最好不要修改系统环境变量。在用户区，我们可以设置与 Vim 相关的环境变量，并保持它们在不同登录会话间的持续存在。

UNIX/LINUX的Bash与其他Bourne shell

编辑适当的shell配置文件（例如Bash用户应编辑`.bashrc`）并插入如下行：

```
VARABC=somevalue
VARXYZ=someothervalue
MYVIMRC=myfavoritevimrcfile
export VARABC VARXYZ MYVIMRC
```

上述各行的顺序没有关系。`export`语句只让变量被shell中运行的程序看到，因而把变量转变为环境变量。在输出前后均可设置变量的值。

UNIX/LINUX C shells

编辑适当的shell配置文件（例如`.cshrc`）并插入如下行：

```
setenv VARABC somevalue
setenv VARXYZ someothervalue
setenv MYVIMRC myfavoritevimrcfile
```

关系到Vim的环境变量

接下来列出大多数Vim的环境变量及其影响。

Vim的`-u`命令行选项覆盖Vim的环境变量，并直接写入指定的初始化文件。`-u`不会覆盖非Vim 的环境变量：

SHELL

指定Vim用于执行shell命令（!!、:!等）的shell或外部命令解释器。在MS-DOS中，如果未设置SHELL，则改用COMSPEC环境变量。

TERM

设置Vim的内部internal term选项。但该选项其实不太需要，因为编辑器会以自己觉得适当的方式设置其终端。换句话说，Vim大概比一个预定义的变量更了解终端的需求。

MYVIMRC

覆盖Vim对初始化文件的搜索。如果启动时找到MYVIMRC的值，Vim即假设这个值是个初始化文件的名称；如果文件存在，则从中取得初始化设置。不再咨询其他文件（请参考前一节的搜索顺序）。

VIMINIT

指定Vim启动时欲执行的ex命令。在命令间以竖线（|）分隔即可定义多个命令。

EXINIT

与VIMINIT相同。

VIM

包含标准Vim安装信息的系统目录路径（只包含信息，而不是被Vim使用）。

注意：如果机器上不只安装了一种Vim，VIM很可能依据用户启动的版本显示不同的值。例如在某位用户的机器上，Cygwin设置VIM环境变量为/usr/share/vim，但vim.org 包则可能把它设为C:\Program Files\Vim。

在更改Vim文件时，系统路径很重要，若是写错文件路径的话，更改或许不会发生作用！

VIMRUNTIME

指向Vim支持文件，例如在线说明文档、语法定义以及插件目录。Vim通常能自己找出这些文件。如果用户设置了这个变量——例如设置成vimrc文件，则有可能在安装新版本Vim时造成错误，因为用户的个人VIMRUNTIME变量或许指向旧的、不存在的或不合适的地址。

新的移动命令

Vim 提供所有vi移动或运动命令，其中大多数已列于第三章，我们再加上一些命令，一起列于表10-1。

表10-1：Vim 的移动命令

命令	说明
<C-End>	转至文件的末端，例如文件最后一行的最后一个字符。如果加上数字，则以数字指定行，跳至该行的最后一个字符
<C-Home>	转至文件第一行的第一个非空格字符。它的行为与<C-End>不一样，<C-Home>不会把光标移到空格上
count%	转至依据文件百分比计算出的行，光标置于该行附近第一个非空白的行。有个重点值得注意，Vim以文件的行数为计算根据，而不是以总字符数量为根据。这乍听之下似乎不怎么重要，但假设有个包含200行的文件，它的前195行各包含5个字符（例如$4.98这类价钱），但最后4行包含1 000个字符。在Unix中，计算字符时，此文件大约包含： (195 * (5 + 1))（*前面只有5个字符的行的字符数*） + 2 + (4 * (1 000 + 1))（*有1 000个字符的行的字符数*） 或说5 200个字符。真正的50%的计算量应该落在第 96 行，但Vim 的50%移动指令却会把光标放在第100行
:go n n go	转至缓冲区的第n个字节。所有字符包括行末字符都计算在内

可视模式的移动

Vim让用户能以可视化的方式定义所选项，且能在此可视化所选项上执行编辑命令。这个功能与在图形化编辑器中点击并拖动鼠标造成的高亮效果很像。Vim的可视模式提供了确认工作执行范围的便利性，而且所有具有此功能的Vim命令都可在所见的已选择文本上工作。这项功能让我们能在较不复杂的编辑器里对高亮的文本执行比传统剪切与复制更为复杂的工作。

我们能在Vim里选择一个可视区域，就像其他编辑器采取点击并拖动鼠标而形成的效果一样。但Vim在定义这种可视选择项时，也能让我们使用它的其他命令以及一些特殊的可视模式命令。

举例而言，我们可以在正常模式中输入v以进入可视模式。进入可视模式后，任何移动光标的命令除了移动光标到新位置，沿途还会以高亮显示文本。所以，“下个词”（w）命令在可视模式中，除了把光标移动到下个词，还会以高亮显示选择的文本。多一次移动，即可根据移动范围适当地扩展选择项。

在可视模式中，Vim使用一些特有命令，方便我们借由选择光标附近的文本对象而扩展所选择文本。例如，光标可能位于某个“词”中，同时也位于某个“句子”中，还位于

某个“段落”中。Vim让我们以扩展高亮区域至文本对象的命令来新增到可视选择项。想以可视化方式选择词汇时，可使用aw（当你在可视模式中时）。

Vim使用表10-2中的移动命令，以利用“可视模式”的优势。在此模式中，高亮显示的行与字符均放入缓冲区，以便提供可视提示，了解Vim后续动作的目标文本。我们有数种方式可以强调缓冲区里的高亮可视区域。在基于文本的模式中，键入v即可打开或关闭可视模式。打开时，于光标移动时选择并高亮显示缓冲区。在gvim中，则只需点击并拖动鼠标，即可选择所需区域。如此即可设置Vim的可视标志。

表10-2列出一些Vim的可视模式中的移动命令。

表10-2：Vim 的可视模式移动命令

命令	说明
*count*aw, *count*aW	选择*count*所指定的单词数量。分隔单词的空格不视为一个词，这一点与iw的计算方式不同（请参考下一项）。小写w把标点符号也视为一个词；大写W则只以空格作为单词的分隔
*count*iw, *count*iW	选择*count*所指定的单词数量。分隔单词的空格也视为一个词。小写w把标点符号也视为一个词，大写W则只以空格作为单词的分隔
as, is	增加选择一个句子，或增加选择内部句子（inner sentence，不含空格）
ap, ip	增加选择一个段落，或增加选择内部段落（inner paragraph）

想进一步了解文本对象的细节内容及文本对象在可视模式中的应用，请善用帮助命令：

```
:help text-objects
```

扩展的正则表达式

在所有vi同类品中，Vim提供最丰富的正则表达式匹配工具。下面列出的说明大部分都出自Vim的说明文档：

\|

表示交替（alternation），house\|home。

\+

它前面的正则表达式需被匹配一次或多次。

\=

它前面的正则表达式需被匹配零次或一次。

\{*n*,*m*}

它前面的正则表达式需被匹配*n*至*m*次，采取贪多原则，匹配越多次越好。*n*与*m* 是0到32 000间的数值。Vim只需要在左花括号前标示反斜线，而不是在右花括号。

\{*n*}

它前面的正则表达式需被匹配*n*次。

\{*n*,}

它前面的正则表达式需至少被匹配*n*次，匹配越多次越好。

\{,*m*}

它前面的正则表达式需被匹配0至*m*次，匹配越多次越好。

\{}

它前面的正则表达式需被匹配零或多次，匹配越多次越好（与*相同）。

\{-*n*,*m*}

它前面的正则表达式需被匹配*n*至*m*次，但匹配最低次数即可。

\{-*n*}

它前面的正则表达式需被匹配*n*次。

\{-*n*,}

它前面的正则表达式需至少被匹配*n*次，但匹配最低次数即可。

\{-,*m*}

它前面的正则表达式需被匹配0至*m*次，但匹配最低次数即可。

\i

匹配任何标识符的字符，标识符由isident选项定义。

\I

与\i大致相同，但排除数字。

\k

匹配任何关键字的字符，关键字由iskeyword选项定义。

\K

与\k大致相同，但排除数字。

\f

匹配任何文件名称的字符，文件名称由isfname选项定义。

\F

与\f大致相同，但排除数字。

\p

匹配任何可打印的字符，由isprint选项定义。

\P

与\p大致相同，但排除数字。

\s

匹配一个空格字符（需确实为空格或tab字符）。

\S

匹配任何不为空白或tab字符的内容。

\b

退格键（backspace）。

\e

Escape键。

\r

回车键（carriage return）。

\t

Tab。

\n

为未来的使用而保留。它将用于匹配多行的模式。请参考Vim的说明文档以了解更多细节。

~

匹配前一个替换字符串。

\(...\)

为*、\+、\=提供分组，如匹配在替换命令的替换部分中的模式子文本（如 \1、\2 等等）。

\1

匹配一组字符串，该字符串匹配第一组\(与\)中的子表达式。例如，\([a-z]\).\1 匹配*ata*、*ehe*、*tot*……\2、\3……及更多或许用于表示第二、三组子表达式。

isident、iskeyword、isfname、isprintoptions选项分别定义被视为标识符、关键字、文件名称、可打印的字符。利用这些选项，可让正则表达式的匹配非常灵活。

自定义可执行文件

对多数用户而言，Vim的默认值已经非常足够。时下的计算机对全功能的Vim可执行文件已可提供充足的处理能力（内存与处理周期）。我们可取用所有的Vim扩展功能，且确信其性能表现良好。然而，在某些状况或环境下，或许需要功能较简洁的Vim。

有些用户或许需要Vim占用最小的空间，例如在内存有限的手提设备上运行Linux 时。用户或许也用不到编译的功能，例如拼写检查（因为用户可能是程序员，对于模拟字处理器的功能不感兴趣）或perl（因为perl或许未安装在他们的机器上）。

接受可得的功能，比起使用新选项重新配置、重新编译与重新安装Vim，实在简单多了。



第十一章

Vim的多窗口功能

Vim默认在一个窗口中编辑所有文件，在文件间移动或者移动到某个文件的不同部分时一次只显示一个缓冲区。但Vim也提供了多窗口编辑的功能，可简化复合的编辑任务。多窗口与在图形终端上启动多个Vim实例不一样。本章讨论如何在一个运行中的Vim进程里（后文将称为会话〔*session*〕）使用多窗口。

开始编辑会话时即可打开多窗口，也可以在工作会话开始后才创建新窗口。你可以一直在编辑会话中增加新窗口，加到自己都无法分辨再动手清除，直到只剩下一个编辑窗口。

我们举一些多窗口能简化工作的例子：

- 编辑许多需要采取相同格式的文件时，一定想从视觉上先行比较，再分别编辑
- 可在许多文件或某个文件的多个部分间快速地重复文本的剪切与粘贴
- 显示文件的某个部分供参考，以协助同一文件中其他部分的工作
- 比较同一个文件的两个版本

Vim提供许多管理窗口的便利功能，包括：

- 水平或垂直地分割窗口
- 快速在窗口间前进与后退的导航能力
- 复制并移动文本到许多个窗口，或在多个窗口间复制并移动文本
- 窗口的移动与重新安排位置
- 工作时搭配缓冲区，包括隐藏缓冲区（将在稍后说明）
- 与外部工具一同使用多窗口，例如diff命令

在本章中，我们将引导各位体验多窗口。包括如何启动一个多窗口会话，讨论这类编辑

会话的功能与诀窍，并讨论如何在退出时确保你的所有操作都妥善地保存（或于需要时妥善地丢弃窗口！）讨论主题如下：

- 多窗口编辑工作的初始化或启动
- 多窗口的:ex命令
- 在窗口间移动光标
- 在显示区中移动窗口
- 调整窗口尺寸
- 缓冲区及其与窗口的交互
- 分页编辑（就像现在的浏览器所提供的分页浏览与对话框）
- 关闭与离开窗口

启动多窗口编辑

我们可于打开Vim时启动多窗口编辑，亦可于编辑会话中分割窗口。Vim的多窗口编辑是动态的，可让我们打开或关闭窗口，并可于任何时间点在窗口间移动，至少大多数情境下均如此。

从命令行（shell）启动多窗口

默认情况下，Vim只为一个会话打开一个窗口，即使打开时已指定多个文件也是如此。虽然我们不确定Vim为何不在指定多个文件时打开多个窗口，但或许是因为使用一个窗口才与vi的行为一致。多个文件要占多个缓冲区，每个文件都有自己的缓冲区（很快后面将讨论到缓冲区）。想从命令行打开多个窗口，请使用Vim的-o选项。例如：

```
$ Vim -o file1 file2
```

上例打开一个编辑会话，其显示为水平分割成两半的窗口，一个文件使用一个窗口（见图11-1）。Vim会试着为命令行上列出的每个文件打开一个编辑窗口。如果画面分割后不能容纳所有文件窗口，则命令行上的第一个文件将取得窗口；余下的文件则载入缓冲区，用户不能看到（但仍可取用）。

另一种命令行预分配窗口的形式，则是在-o后附加数量*n*：

```
$ Vim -o5 file1 file2
```

上例打开一个编辑会话，屏幕被水平分割成5个相同尺寸的窗口，最顶端的窗口包含file1，第二个窗口则包含file2（见图 11-2）。

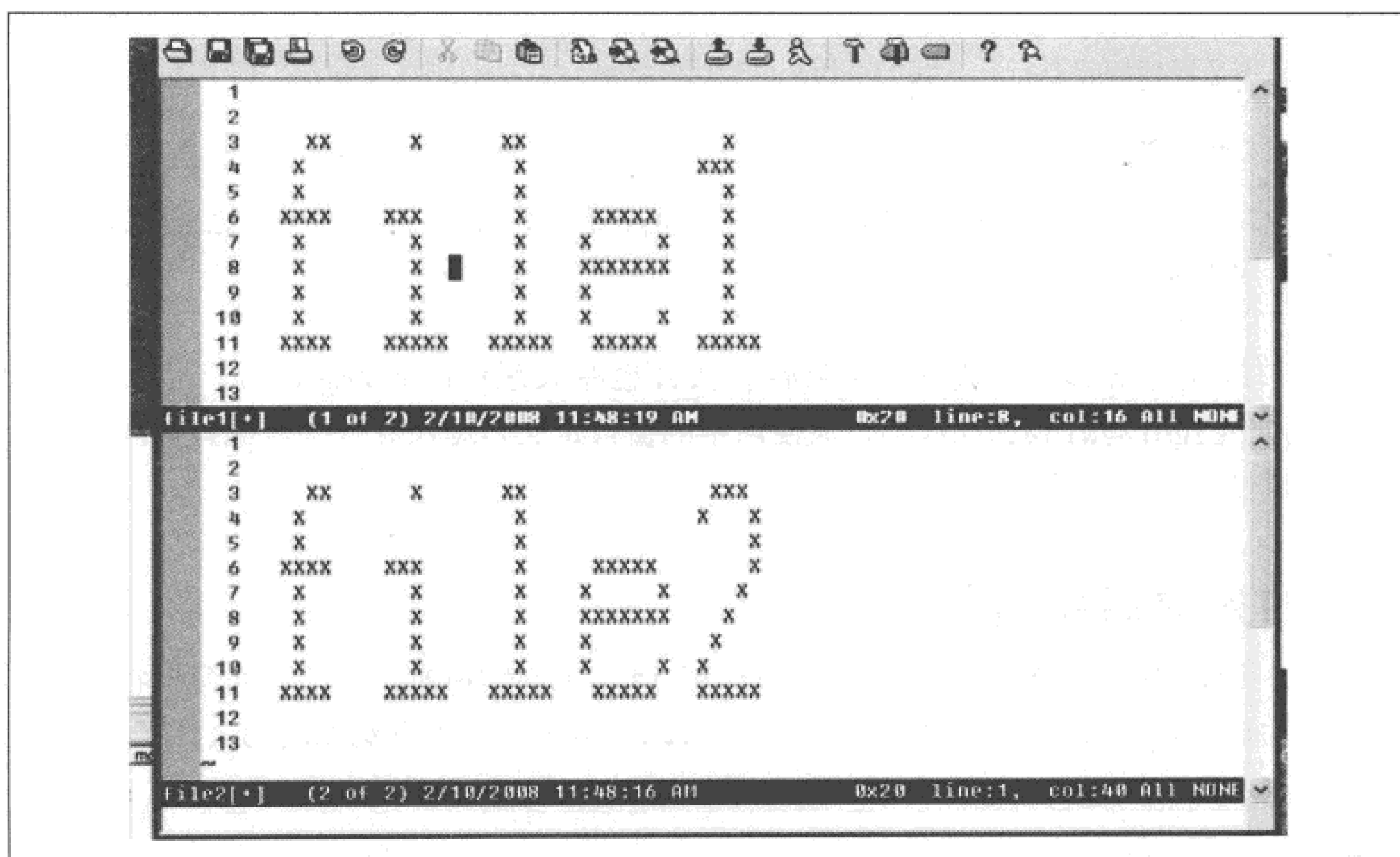


图11-1： “Vim-o file1 file2”的结果

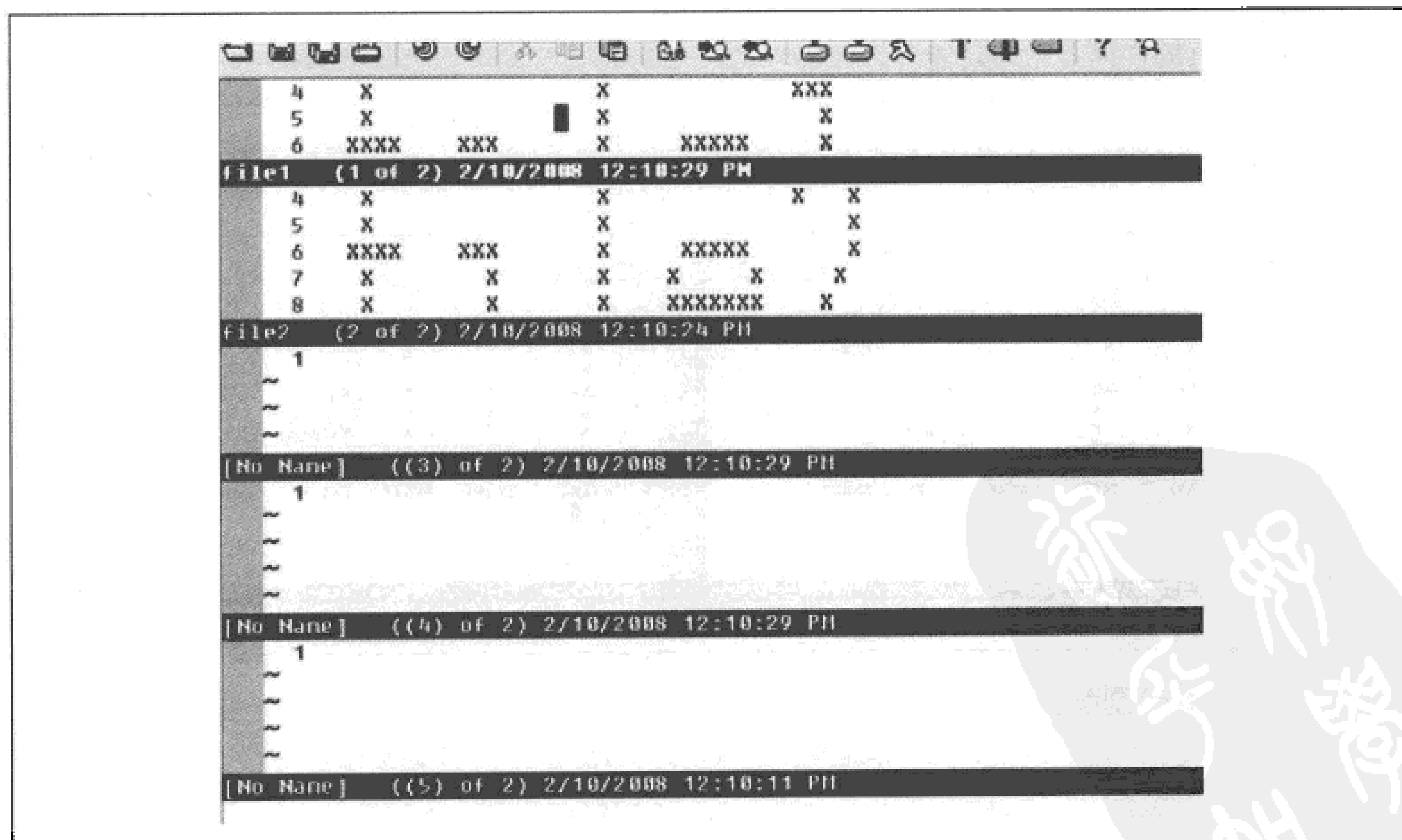


图11-2： “Vim-o5 file1 file2”的结果

注意： 当Vim创建超过一个窗口时，它默认为每个窗口创建状态行（至于处理一个窗口时，则默认不呈现任何状态行）。我们可用Vim的laststatus选项控制这项行为，如下所示：

```
:set laststatus=1
```

设置laststatus为2，即可看到每个窗口的状态行，就算只有一个窗口也没问题（最好在.vimrc文件中设置）。

因为窗口的尺寸会影响可读性与可用性，或许也有人想控制Vim的窗口尺寸。可使用Vim的winheight与winwidth选项，为使用中的窗口定义合理尺寸（其他窗口可能会顺应要求也调整尺寸）。

Vim的多窗口编辑

于Vim中可以启用并调整窗口配置。使用:split即可创建新窗口。当前的窗口将分割成两半，均显示相同缓冲区的内容。这样我们可以在两个窗口中浏览相同的文件了。

注意： 本章提到的命令，许多都有便捷的按键序列。例如，^Ws可分割窗口（所有Vim的窗口相关命令均以^W开始，这有助于记得它是“窗口（window）”命令）。为了讨论方便，我们只列出命令行的方式，因为它们提供了可选参数（可自定义缺省行为）的额外功能。如果各位发现自己经常使用的命令，你可以在Vim说明文档中轻松地找到相应按键序列，可参考第163页的“内置帮助功能”。

相似地，也可以创建全新的垂直分割编辑窗口，此时使用:vsplit命令（见图 11-3）。

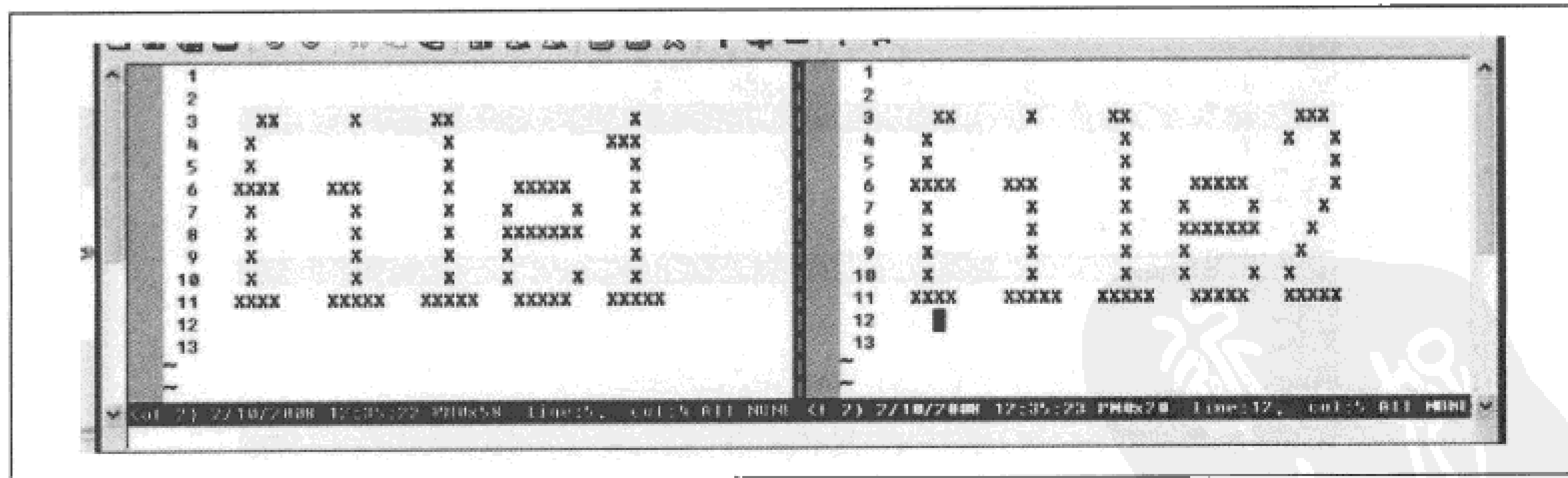


图11-3：垂直地分割窗口

Vim以上述方式（水平或垂直地）分割窗口，且:split命令行中未指定文件时，我们将于两个窗口中编辑同一份文件。

注意：不相信你在同时编辑同一个文件吗？分割你的编辑窗口并滚动分割后的窗口，它们会呈现文件中的相同内容。请挑一个窗口做点改变，然后看看另一个窗口。多神奇啊！

这项功能为何有用？又该如何使用？本章作者运用分割窗口的方式，经常是在编写shell脚本或C程序时，编写一个描述程序用途的代码块（一般而言，程序将送入--help选项时呈现此块）。我分割了显示画面，用一个窗口列出用途说明，把这个窗口当成编辑另一个窗口中的代码的样板；而代码窗口中的工作是解释用途说明窗口里讲到的所有选项与命令行参数。这类代码通常（几乎总是）很复杂，结束时与用途说明距离太远，因此不能在一个窗口里呈现所需的一切东西。

如果你想编辑或浏览另一个文件，而又不想失去在当前文件中的编辑位置，请以新文件为参数传给:split命令。例如：

```
:split otherfile
```

下一节将详细说明窗口的分割与取消分割。

打开窗口

本节深入讨论如何在分割窗口时取得理想的精确行为。

新窗口

前面讲过，打开新窗口最简单的方式就是使用:split（水平分割）或:vsplit（垂直分割）。接下来将更深入地讨论众多相关命令及变化。我们也会讲到命令语法，以供快速参考。

分割窗口的选项

完整的打开新的水平窗口的:split命令如下所示：

```
:[n]split [++opt] [+cmd] [file]
```

命令中：

n

为Vim指定在新窗口中显示的行数，新窗口位于画面顶端。

opt

传递Vim选项信息给新的窗口会话（请注意，它的前面必须加上两个加号）。

cmd

传入欲在新窗口中执行的命令（请注意，它的前面必须加上一个加号）。

file

指定欲在新窗口中编辑的文件。

假设你正在编辑文件，现在想分割窗口以编辑名为otherfile的另一个文件。另外，还想确保会话使用unix的fileformat（用于确认以line feed作为每行结尾，而不是使用carriage return和line feed的组合）。最后，想使窗口有15行高。请输入：

```
:15split ++fileformat=unix otherfile
```

想简单地分割屏幕，呈现相同的文件于两个窗口，并使用所有当前的默认值，你可以使用按键命令^Ws、^WS、^W^S。

注意： 如果想让窗口总是平均地分割，请设置equalalways选项，最好设置在你的.vimrc文件中，让设置能在不同会话间获得保存。默认情况下，设置equalalways可让窗口的水平或垂直分割均维持相同尺寸。加入eadirection选项（其中hor、ver或both分别代表水平、垂直或两者皆有）可控制该平均分割的方向。

如下的:split命令可如前所述打开一个水平窗口，但有一点细微差异：

```
:[n]new [++opt] [+cmd] [file]
```

除了创建新窗口，自动命令WinLeave、WinEnter、BufLeave、BufEnter也被执行了（关于自动命令的更多信息，请参考第208页的“自动命令”一节。）

除了水平分割命令，Vim也提供类似的垂直分割命令。例如想垂直地分割窗口时，不是使用:split或:new，而是改用:vsplit或:vnew。

但有两个水平分割命令没有垂直分割版的对应命令：

:sview *filename*

水平分割屏幕以打开新窗口，并为缓冲区设置 readonly。:sview中的 *filename* 为必要参数。

:sfind [++*opt*] [+*cmd*] *filename*

与:split运作方式相近，但在path中寻找*filename*。如果Vim未找到文件，则不会分割窗口。

有条件的分割命令

Vim可让我们执行一种命令，在找到新文件时才打开窗口。:topleft *cmd*告诉Vim：执行*cmd*并显示一个新窗口，如果*cmd*成功打开了新文件，光标需在左上角。这个命令可能产生三种不同结果：

- *cmd*水平分割了窗口，新窗口占据Vim窗口的上半部。
- *cmd*垂直分割了窗口，新窗口占据Vim窗口的左半部。
- *cmd*未分割窗口，而是把光标移到当前窗口的左上角。

窗口命令总结

表11-1总结了分割窗口的命令。

表11-1：窗口命令总结

ex命令	vi命令	说明
:[n]split [++*opt*] [+*cmd*] [*file*] ^W^S	^Ws ^WS	水平分割当前窗口成两个窗口，并把光标置于新窗口。可选的*file*参数可把指定文件放入新创建的窗口。分割的两个窗口会尽可能地尺寸相同，具体根据余下的窗口空间而定
:[n]new [++*opt*] [+*cmd*]	^Wn ^W^N	与split相同，但会打开一个新窗口，编辑空白文件。请注意，缓冲区不会有名称，除非我们另外指定
:[n]sview [++*opt*] [+*cmd*] [*file*]		只读版的:split
:[n]sfind [++*opt*] [+*cmd*] [*file*]		分割窗口并于新窗口打开文件（如果指定了*file*）。于path中寻找*file*
:[n]vsplit [++*opt*] [+*cmd*] [*file*]	^Wv ^W^V	垂直分割当前窗口成两个窗口，并于新窗口打开文件（如果指定了*file*）
:[n]vnew [++*opt*] [+*cmd*]		垂直分割版的:new

游走窗口间（在窗口间移动光标）

使用gvim与Vim时，在窗口间移动其实很简单。gvim已默认支持鼠标的点击，至于 Vim

则可打开mouse选项。Vim的默认设置:set mouse=a的效果不错，为所有用途——命令行、输入、导航都激活鼠标的使用。

如果你没有鼠标，或者比较喜欢用键盘控制会话，Vim则提供全套导航命令，可在会话窗口间快速而准确地移动。Vim的按键组合里也一致以易于记忆的^W用于窗口间的移动。接下来的按键组合定义了光标的移动或其他行动，有经验的vi与Vim用户均应熟记这些组合，因为它们紧密地对应编辑时的行动命令。

与其解释每个命令及其行为，我们采用范例说明，再看命令概要表，应该能更能一目了然。

想从当前的窗口移动到下个窗口，请键入CTRL-W j（或CTRL-W <down>、CTRL-W CTRL-J）。CTRL-W表示“窗口（windows）”命令，j则与Vim的j命令类似，用于移动光标至下一行。

表11-2整理了窗口导航命令。

注意： 就像其他 Vim与vi命令一样，只要在这些命令前加上数量，就可执行多次。例如，3^Wj告诉Vim，从当前窗口跳到往下数的第三个窗口。

表11-2：窗口导航命令

命令	说明
CTRL-W <DOWN>	移到下一个窗口
CTRL-W CTRL-J	请注意，这个命令最后不会回到最上面的窗口，只是单纯地移到下一个窗口。如果光标位于屏幕最底端的窗口，本命令即无效。还有，本命令在“往下移动”时会跳过同一行的其他窗口。假设在当前窗口的右边还有一个窗口，本命令不会跳到旁边（右边）的窗口（请用CTRL-W CTRL-W在窗口间循环移动）
CTRL-W <UP>	移动到上一个窗口。与CTRL-W j命令的移动方向刚好相反
CTRL-W CTRL-K	
CTRL-W k	
CTRL-W <LEET>	移动到位于当前窗口左边的窗口
CTRL-W CTRL-H	
CTRL-W h	
CTRL-W <BS>	

表11-2：窗口导航命令（续）

命令	说明
CTRL-W <RIGHT>	移动到位于当前窗口右边的窗口
CTRL-W CTRL-L	
CTRL-W l	
CTRL-W w	移动到下方的窗口或右边的窗口。请注意，这个命令与CTRL-W j不一样，这个命令会在所有的Vim窗口间循环移动。到达最底端的窗口后，又会重新移到最左上角的窗口，开始新一轮循环
CTRL-W CTRL-W	
CTRL-W	移动到上方或左边的窗口。与CTRL-W w命令的移动方向刚好相反
CTRL-W t	移动光标到最左上角的窗口
CTRL-W CTRL-T	
CTRL-W b	移动光标到最右下角的窗口
CTRL-W CTRL-B	
CTRL-W p	移动到前一个（最后访问的）窗口
CTRL-W CTRL-P	

帮助记忆的诀窍

t与b分别是顶端（top）与底端(bottom)窗口的快速记忆方式。

为了保持大写与小写实现相反的惯例，w在窗口间移动的方向与CTRL-W W相反。

Control字符不会区分大小写。换句话说，在按下shift键的同时按住CTRL-键并无效果。不过，其后输入的键仍有大小写的差别。

移动窗口

在Vim中有两种移动窗口本身的方式。一种只是简单地在屏幕上切换窗口；另一种则是改变窗口的实际布局。第一种情况中，虽然窗口在屏幕上的位置有变，但尺寸维持不变；第二种情况中，窗口不只移动，还调整其尺寸，以填充它们移向的位置。

移动窗口本身（轮换或交换）

移动窗口但不会调整布局的命令共有三个。其中有两个依特定方向轮换窗口的位置（向右下或向左上轮换），还有一个则交换两个可能并未毗邻的窗口的位置。这些命令的对象只限当前窗口所在的行或列上。

CTRL-W r向右或向下方轮换窗口。与它相对的命令为CTRL-W R，轮换方向相反。

关于窗口的轮换，我们可将Vim窗口的一列或一行想成一个一维数组。CTRL-W r把数组里的每个元素向右移动一位，最末端的元素则移到空出来的第一位。CTRL-W R则是把元素往另一个方向移动。

如果没有其他窗口与当前窗口同列或同行，则这些命令不会有动作。

在Vim轮换窗口后，光标仍一直在执行轮换命令的窗口里。也就是说，光标随着窗口移动。

CTRL-W x与CTRL-W X能交换同列或同行的窗口的位置。Vim默认为交换当前窗口与它的下一个窗口的位置；如果下方没有窗口，则试着与上一个窗口交换位置。亦可在此命令前加上数量，与指定的窗口交换位置。例如，要与当前窗口往下数的第三个窗口交换位置，命令应写成：3^Wx。

与前面提到的两个命令类似，光标一直在执行交换命令的窗口里。

移动窗口并改变其布局

有5个负责移动并回流（reflow）窗口的命令：有两个命令可以移动当前窗口至最顶端或最底端（且使用屏幕的全部宽度）；有两个命令可以移动当前窗口至最左端或最右端（且使用屏幕的全部高度）；第五个命令则把当前的窗口移到另一个现有的分页（请参考第195页的“分页编辑”一节）。前四个命令与其他Vim命令具有相似的易记性。例如CTRL-W K对应了以k键作为“向上”的传统概念。表11-3整理了这些命令。

表11-3：移动并回流窗口的命令

命令	说明
^WK	移动当前窗口至屏幕顶端，使用屏幕的全部宽度
^WJ	移动当前窗口至屏幕底端，使用屏幕的全部宽度
^WH	移动当前窗口至屏幕左端，使用屏幕的全部高度
^WL	移动当前窗口至屏幕右端，使用屏幕的全部高度
^WT	移动当前窗口至新的现有分页

实在很难描述这些布局命令的精确行为。在移动并扩展窗口为满屏的宽度或高度后，Vim重新以合理的方式分配其他窗口。重新分配的行为亦可受到一些窗口选项设置的影响。

窗口移动命令概要

表11-4与11-5整理了本节介绍的命令。

表11-4：轮换窗口位置的命令

命令	说明
^Wr	向右或向下轮换窗口
^W^R	
^WR	向左或向上轮换窗口
^Wx	与下一个窗口交换位置；加入数量*n*时，则与向下数的第*n*个窗口交换位置
^W^X	

表11-5：改变位置与布局的命令

命令	说明
^WK	移动窗口至屏幕顶端并使用全部宽度。光标一直在移动的窗口里
^WJ	移动窗口至屏幕底端并使用全部宽度。光标一直在移动的窗口里
^WH	移动窗口至屏幕左端并使用全部高度。光标一直在移动的窗口里
^WL	移动窗口至屏幕右端并使用全部高度。光标一直在移动的窗口里
^WT	移动窗口至新分页。光标一直在移动的窗口里。如果当前的窗口是当前分页里的唯一窗口，则不会发生任何操作

调整窗口尺寸

大家已经比较熟悉Vim的多窗口功能，想必对这些功能需要多一点控制。本节说明如何改变当前窗口的尺寸，当然这也会影响到屏幕上的其他窗口。在使用窗口分割命令时Vim提供了控制窗口尺寸与调整尺寸的选项。

如果你不想用命令控制窗口尺寸，请使用gvim并以鼠标调整。用鼠标简单地点击及拖动窗口边界，即可调整窗口尺寸。遇到垂直分割的窗口，用鼠标点击 |字符（垂直分隔符）。水平分割的窗口以状态行分隔。

窗口尺寸调整命令

大家可能猜到了，Vim有垂直与水平的调整命令。就像其他窗口命令，调整命令也以CTRL-W开始，并与有助记忆的设备映射良好，让它们易学易记。

CTRL-W =试图调整所有窗口至相同尺寸（此命令受到当前的winheight与windwidth值影响，后续章节将有讨论）。如果可用的屏幕块不能平均划分，Vim也会试着尽可能地平均分割。

CTRL-W -将当前窗口的高度减少一行。Vim还有一个ex命令，能明确地指定欲减少的窗口尺寸。例如，`resize -4`即可从当前窗口里减去4行，并把减去的行数分配给它上面的窗口。

注意： 有件事很有趣，即使不在一个多窗口编辑会话中，Vim也会减小窗口尺寸。乍看之下有点违反常理，附加效果是Vim会减小窗口面积，空出来的屏幕面积都留给命令行窗口。通常，命令行窗口一般只使用一行，但有时候也需要使用高于一行的命令行窗口（我们所知的最常见理由是提供足够空间，让Vim呈现完整的命令行状态与反馈消息，而不需提示符号从中插手）。也就是说，最好使用`:resize`命令调整当前窗口的尺寸，winheight选项则用于调整命令行窗口。

CTRL-W +将当前窗口的高度增加一行。`:resize +`*n*命令则可为当前的窗口增高*n*行。一旦到达窗口的最大高度，再使用这个命令，也不会有效果。

注意： 本书作者最喜欢的CTRL-W +与CTRL-W -的使用方式，是把这两组命令映射至自定义按键，而且按键最好紧临。+键是个方便的选择。虽然它是Vim的“向上”命令，但对于熟练的Vim用户而言，只是一个重复的、多余的行为（熟练的人会改用k命令）。因此很适合映射其他行为，本例映射到CTRL-W+。在键+旁边是-，但因为打出-不用按Shift键，而+需要，我们改用按Shift键才会出现的-键映射至CTRL-W -。现在你有两个毗邻的快捷按键，可以轻易而快速地扩大或缩小当前窗口了（以水平分割而言）。

`:resize `*n*设置当前窗口的水平尺寸为*n*行。设置值为绝对尺寸，与前面描述的命令设置相对尺寸的行为不一样。`z`*n*设置当前窗口的高度为*n*。请注意*n*并非可选项，省略它将变成vi/Vim的z命令，用于把光标移到屏幕顶端。

CTRL-W <与CTRL-W >分别能减少与增加窗口的宽度。想想一些常见设备上的<<与>>，有助于记忆这些命令与对应功能。

最后，CTRL-W |可调整当前窗口至可能的最大宽度（默认值）。也可通过`vertical resize `*n*明确指定如何改变窗口。*n*定义了窗口的新宽度。

窗口尺寸调整选项

有些Vim选项能影响上节所述的尺寸调整命令的行为。

在窗口变为“活动中”（active）时，winheight与winwidth分别定义窗口的最小高度与宽度。例如，屏幕可容纳两个相同尺寸的窗口（45 行），默认的Vim行为即为平均分配窗口的尺寸。如果winheight的设置值大于45行——例如60行，则Vim将把每次光标移向的窗口调整为60行，另一个则调整为30行。在同时编辑两个文件时，这种行为蛮方便的：在切换窗口、切换文件时自动增加窗口尺寸，以提供最大上下文范围。

equalalways让Vim在分割或关闭窗口后，把窗口调整为相同尺寸。它是个在增加及删除窗口时都维持相等的窗口分配尺寸的好选项。

eadirection定义equalalways的方向权限，可使用的值包括hor、ver、both，分别是在水平、垂直、两个方向上都调整窗口尺寸为相等的。每次分割或删除窗口都会运用尺寸的调整。

cmdheight设置命令行的高度。稍早提过，在只有一个窗口时，减小窗口的高度将增加命令行的高度。使用此选项，即可维持命令行的高度。

最后两个选项，winminwidth与winminheight，用于确定窗口调整时的最小宽度与高度。Vim把这两个选项值视为硬性规定，所以窗口尺寸永远不许小于这两个值的规定。

尺寸调整命令概要

表11-6整理了调整窗口尺寸的方式，选项以:set命令设置。

表11-6：窗口尺寸调整命令

命令或选项	说明
^W=	重新调整所有窗口至相同尺寸。当前窗口将以选项 winheight与winwidth的设置为准
:resize -*n* ^W-	减少当前窗口的尺寸。默认减少量为一行
resize +*n* ^W+	增加当前窗口的尺寸。默认增加量为一行
:resize *n* ^W^_ ^W_	设置当前窗口的高度。默认为最大窗口高度（除非指定*n*的值）
z*n* <ENTER>	设置当前窗口的高度为*n*

表11-6：窗口尺寸调整命令（续）

命令或选项	说明
^W<	增加当前窗口的宽度。默认增加量为一栏
^W>	减少当前窗口的宽度。默认减少量为一栏
:vertical resize *n*	设置当前窗口的宽度。默认值为最大窗口宽度
^W|	
winheight选项	进入或创建窗口时，设置其高度至少等于指定的值
winwidth选项	进入或创建窗口时，设置其宽度至少等于指定的值
equalalways选项	窗口数量改变时(无论是因为分割或关闭)，把改变后剩余的窗口调整为相同尺寸
eadirection选项	定义Vim是在垂直、水平还是两个方向上调整窗口尺寸
cmdheight选项	设置命令行高度
winminheight选项	定义最小窗口高度，使用在所有创建的窗口上
winminwidth选项	定义最小窗口宽度，使用在所有创建的窗口上

缓冲区及其与窗口的交互

Vim使用缓冲区作为工作对象的容器。完全了解缓冲区乃是必需技能，有很多控制缓冲区以及在缓冲区中移动的命令。然而，先熟悉一些缓冲区的基本知识并了解缓冲区如何与为何存在于整个Vim会话里，将是值得的。

打开几个窗口以编辑不同文件，是了解缓冲区的不错起点。例如，打开Vim ，编辑file1，然后在这个会话内接连发出命令:split file2与:split file3。现在应该已有三个Vim窗口，各自显示不同的文件。

现在使用命令:ls、:files、:buffers列出缓冲区。你应该看到三行，每一行都有编号并列出文件名称及一些额外信息。这些是Vim分配给本会话的缓冲区。每个文件对应一个缓冲区，每个缓冲区则有独一无二、不会改变的编号。本例中，file1在一号缓冲区，file2在二号缓冲区，依此类推。

如果在任何命令后附加感叹号，还能列出每个缓冲区的额外信息。

在每个缓冲区编号右侧，首先列出的是状态标志（statu flag）。这些标志以表11-7所述的方式描述缓冲区。

表11-7：描述缓冲区的状态标志

状态代码	说明
u	非列出缓冲区。这个缓冲区不会列出，除非使用!。想看到非列出缓冲区（unlisted buffer）的范例，请键入:help。Vim分割当前的窗口，以容纳呈现内置帮助的新窗口。直接使用:ls命令，不会显示帮助使用的缓冲区，需使用:ls!才会列出
%或（相对地）#	%表示当前窗口所用的缓冲区。#则是使用:edit #后跳到的缓冲区
a或（相对地）h	a表示活动中的缓冲区，意为该缓冲区已载入且可见。h则表示隐藏的缓冲区，它虽然存在，但不能在任何窗口中查看
-或（相对地）=	-表示缓冲区把modifiable选项关闭了，这个文件为只读文件。=则表示该文件是不能把状态修改为可调整的只读文件（例如我们没有写入这个文件的系统权限）
+或（相对地）x	+指出缓冲区可以调整。x表示缓冲区具有读入错误

注意： u是种知道正在浏览哪一份Vim帮助文件的有趣方式。例如，在发出:help split命令后，接着发出:ls!命令，将看到一个非列表缓冲区，指向内置的Vim帮助文件--windows.txt。

既然我们已经能列出Vim的缓冲区，现在就于可以讨论缓冲区及其各种应用。

Vim的特殊缓冲区

Vim系统自身使用的一些缓冲区称为特殊缓冲区（special buffer）。例如，上一节提到的帮助缓冲区。一般而言，这些缓冲区不能被编辑或修改。

这里列举4种Vim的特殊缓冲区：

quickfix

包含因为我们的命令（可使用:cwindow查看）或位置列表（location list，可使用:lwindow查看）而创建的错误列表。不可编辑这个缓冲区里的内容！它能帮助程序员重复“编辑—编译—调试”周期。请参考第十四章。

help

包含Vim的帮助文件，于第163页的“内置帮助功能”一节中已经讨论过。:help将这些文本文件放入特殊缓冲区里。

directory

包含目录的内容，也就是某个目录中的文件列表（以及一些有用的额外命令提示）。它是Vim里的快捷工具，能让我们在缓冲区中移动，就像在一般文本文件里

移动，而且可用光标加上ENTER选择欲编辑的文件。

scratch

这些缓冲区包含一般用途的文本。其中的文本可扩充，而且随时可被删除。

隐藏缓冲区

隐藏缓冲区是不显示于任何当前窗口里的Vim缓冲区。如果考虑到多窗口所能占据的屏幕面积，又不想经常下达取得和重写文件的命令，隐藏缓冲区可使编辑多个文件变得较为容易。假设你正在编辑myfile文件，但偶尔又想编辑另一个文件myOtherfile。如果设置了hidden选项，即可用:edit myOtherfile编辑myOtherfile，这时将隐藏myfile的缓冲区，而在它的位置上显示myOtherfile的缓冲区。你可以通过:ls确认这一点，此命令将列出两个缓冲区，而myfile标示为*hidden*。

缓冲区命令

大约有50个专门针对缓冲区的命令。许多命令很有帮助，但大部分都超出本书的讨论范围。Vim自动在打开与关闭多个文件或窗口时管理缓冲区。缓冲区命令几乎能对缓冲区执行所有操作。这些命令通常用在脚本中，以处理卸载、删除、调整缓冲区等任务。

但在平常使用时，最好记住两个缓冲区命令，因为它们能一次对许多文件做许多工作：

windo *cmd*

其是"window do"的简称（至少我们认为它还蛮好记的），这个伪缓冲区命令（它其实是个窗口命令）在每个窗口里执行指定命令*cmd*。它的动作类似于先到顶端窗口（^Wt），而后循环于各个窗口并于其中执行指定命令（如在每个窗口中发出:cmd命令）。它只在当前分页中动作，若遇到任何因为:cmd产生的错误，即停在产生错误的窗口。产生错误的窗口随即成为新的当前窗口。

cmd并未获准改变窗口状态。也就是说，它不能删除、新增或改变窗口顺序。

注意： cmd能使用管道（|）符号串联多个命令。但不要把这种表示方式与Unix shell的管道输送命令的惯例混淆。本处的命令为依序执行，从第一个命令开始在每个窗口逐一执行，接着逐一使用第二个命令于每一个窗口，依此类推。

下面是:windo的范例。假设你正在编辑一组Java文件，但因为某些原因，发现有个类型的名称大小写不正确。你需要改变每个myPoorlyCapitalizedClass为MyPoorlyCapitalizedClass。可以使用:windo这么做：

```
:windo %s/myPoorlyCapitalizedClass/MyPoorlyCapitalizedClass/g
```

很棒吧！

bufdo[!]*cmd*

它是与windo类似的命令，但操作对象是所有编辑会话的缓冲区，而不只是当前分页中的可见缓冲区。bufdo与windo的行为相似，遇到第一个错误立刻中止，并把光标留在命令执行失败的缓冲区里。

下例能更改所有缓冲区为Unix文件格式：

```
:bufdo set fileformat=UNIX
```

缓冲区命令概要

表11-8并不包含所有与缓冲区相关的命令，其中只有本节所述命令的摘要，再加上其他常用命令。

表11-8：缓冲区命令总结

命令	说明
:ls[!] :files[!] :buffers[!]	列出缓冲区与文件名称。如果加上!，则包括非列表缓冲区
:ball :sball	编辑所有参数或缓冲区（sball则会在新窗口中打开它们）
:unhide :sunhide	编辑所有载入的缓冲区（sunhide则会在新窗口中打开它们）
:badd file	把flie加入列表
:bunload[!]	从内存中卸载缓冲区。加上!则可强迫修改过的缓冲区在未保存的状况下被卸载
:bdelete[!]	卸载缓冲区并从缓冲区列表中将其删除。加上!则可强迫修改过的缓冲区在未保存的状况下被卸载
:buffer [*n*] :sbuffer [*n*]	移向缓冲区*n*（sbuffer则会打开一个新窗口）
:bnext [*n*] :sbnext [*n*]	移向接下来的第*n*个缓冲区（sbnext则会打开一个新窗口）
:bNext [*n*] :sbNext [*n*] :bprevious [*n*]	移向后面或前面的第*n*个缓冲区（sbNext与sbprevious则会另开一个新窗口）

表11-8：缓冲区命令摘要（续）

命令	说明
:sbprevious [*n*]	
:bfirst :sbfirst	移到第一个缓冲区（sbfirst则会打开一个新窗口）
:blast :sblast	移到最后一个缓冲区（sblast则会打开一个新窗口）
:bmod [*n*] :sbmod [*n*]	移到第*n*个修改过的缓冲区（sbmod则会打开一个新窗口）

在窗口里追踪标签

Vim把vi标签功能扩展到窗口里，对多窗口提供相同的标签遍历机制（tag traversal mechanism）。使得追踪一个标签也能在新窗口里打开相关联的文件。

标签窗口命令会分割当前的窗口，并追踪标签至匹配标签的文件，或追踪至匹配光标下的文件名的文件。

:stag[!]可分割窗口，以显示找到的标签的位置。包含匹配的标签的新文件成为当前编辑中的窗口，光标则位于匹配的标签上。如果没有找到标签，则命令失败且不会创建新窗口。

注意： 逐渐熟悉Vim的帮助系统后，可试着使用:stag命令分割帮助窗口，而不仅限于同一个窗口中的从一个文件移到另一个文件。

---

^WJ或^W^J能分割窗口并于当前窗口上再打开新的窗口。新窗口变成当前窗口，光标位于找到的匹配标签上。如果没找到与标签匹配的文字，则此命令执行失败。

^Wg] 能分割窗口并于当前窗口上创建新窗口。在新窗口中，Vim执行:tselect *tag*命令，其中*tag*是光标下的标签标识中。如果没有匹配的标签存在，则此命令执行失败。若执行成功，光标将位于新窗口中，新窗口则成为新的当前窗口。

^Wg^J的运作方式与^Wg]大致一样，但它不是执行:tselect，而是执行:tjump。

^Wf（或^W^F）能分割窗口并编辑光标所在处的文件名称。Vim将在可选变量path设置的文件里依序寻找。如果在任何path目录下都找不到文件，则此命令执行失败，也不会打开新窗口。

^WF分割窗口并编辑光标所在处的文件名称。光标将位于编辑该文件的新窗口里，确切位置则是原始窗口中文件名称后列出的编号所指定的行。

^Wgf于新分页中打开光标处的文件。如果文件不存在，则不会创建新分页。

^WgF会于新分页中打开光标处的文件，光标移动到原始窗口中文件名称后列出的编号所指定的行。如果文件不存在，则不会创建新分页。

分页编辑

除了可在多窗口里编辑外，各位知道也可以创建多个分页（tab）吗？Vim允许创建新分页，每个分页各有独自的行为。每个分页中，你可以分割屏幕、编辑多个文件……几乎可执行任何在一个窗口中允许的行为，只是现在你的所有工作都利用分页在一个窗口中被简易地管理。

许多Firefox用户已经非常熟悉并依赖分页浏览的功能，也认识到分页为编辑带来的推动力。对新手而言，分页也是个值得尝试的功能。

我们可在一般的Vim与gvim中使用分页，但gvim会更好用。创建与管理分页的最重要方式包括：

:tabnew *filename*
: 打开新分页并编辑新文件（可选的）。如果并未指定文件，Vim只打开一个新分页并附上空的缓冲区。

:tabclose
: 关闭当前的分页。

:tabonly
: 关闭其他所有分页。如果其他分页中有修改过的文件，则不会移除该分页，除非设置了autowrite选项。此时，所有修改过的文件都在分页关闭前先写入磁盘。

在gvim中，要打开分页只需点击屏幕顶端的分页图标。如果配置了鼠标，也可以在基于字符的终端里使用鼠标打开分页（参考mouse选项）。使用CTRL PAGE DOWN（向右移动一个分页）与CTRL PAGE UP（向左移动一个分页）则可在分页间较容易地左右移动。如果已在最左或最右的分页，而你还要试着向更左边或更右边移动时，Vim则会循环移至相反端的分页。

gvim提供分页的右键弹出菜单，从中可以打开新分页（有没有欲编辑的新文件都可以）与关闭分页。

图11-4是一组分页的范例（请注意分页的右键弹出菜单）。

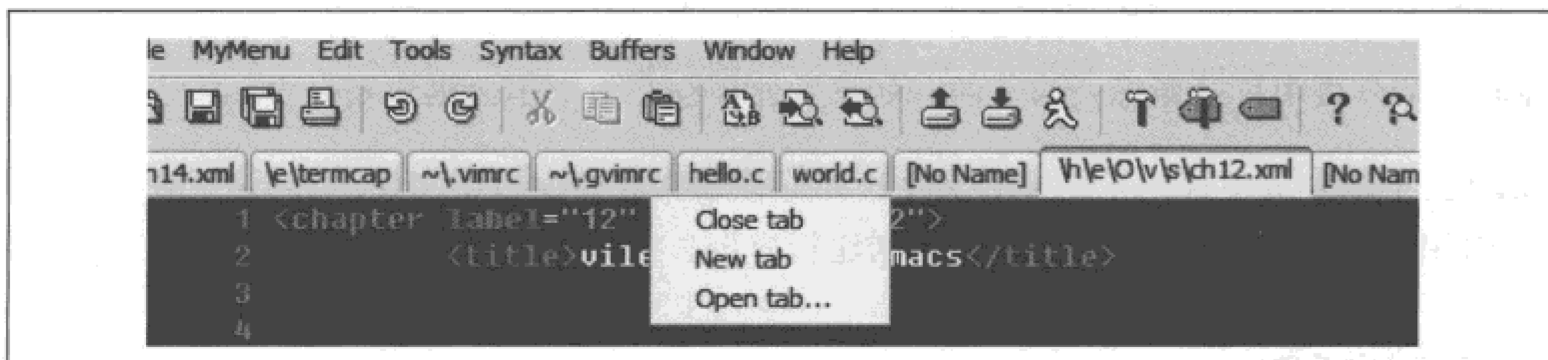


图11-4：gvim分页与分页编辑的范例

关闭与离开窗口

有4种关闭窗口的方式，而且均特别与窗口编辑有关：离开（quit）、关闭（close）、隐藏（hide）、关闭其他窗口。

^Wq（或^W^Q或:quit）其实是窗口版本的:quit命令。它的最简单形式（也就是只有一个窗口的一个编辑会话）就像vi的:quit命令。如果设置了hidden选项，而当前窗口是屏幕上引用文件的最后一个窗口时，则窗口虽被关闭了，但缓冲区仍存在（可被取用）且隐藏。换句话说，Vim仍然存储着文件，我们可以稍后再编辑它。如果并未设置hidden，当窗口为最后一个引用文件的窗口且当前窗口的缓冲区里还有尚未保存的改变时，则命令会失败，以免你所做的改变丢失了。但如果其他窗口显示相同的文件，则会关闭当前的窗口。

^Wc（或:close[!]）关闭当前的窗口。如果设置了hidden选项且关闭的目标是引用文件的最后一个窗口，Vim将关闭窗口并隐藏缓冲区。如果窗口位于分页中而且是该分页的最后一个窗口，则会同时关闭窗口及分页。只要不使用!，这个命令就不会丢弃任何未保存的改变。!告诉 Vim无条件地关闭当前的窗口。

注意：请注意，这个命令并未使用^W^C，因为Vim使用^C作为取消（cancel）命令。因此，如果你试着使用^W^C，^C将取消该命令。

相似地，当^W命令结合^S与^Q一起使用时，有些用户可能会发现终端怎么静止了？因为有些终端把^S与^Q解释为控制字符（control character），用于停止和开始显示信息到屏幕上。如果在使用这些按键组合时发现自己的屏幕莫名地静止了，请改用其他列出的组合再尝试。

---

^Wo、^W^O与:only[!]将关闭当前窗口以外的所有窗口。如果设置了hidden选项，所有被关闭窗口的缓冲区将被隐藏。如果未设置，任何引用到尚未保存改变的文件的窗口将

留在屏幕上。除非使用!，此时所有窗口均会被关闭，文件改变也会被丢弃。这个命令的行为受autowrite选项的影响。如果设置了此选项，所有窗口都会被关闭，但遇到尚未保存改变的窗口时，Vin将在离开前把文件写入磁盘。

:hide [*cmd*]离开当前的窗口并隐藏缓冲区（如果没有其他窗口引用到此缓冲区）。如果提供了可选的*cmd*，则会隐藏缓冲区并执行命令。

表11-9列出了这些命令的总结。

表11-9：用于关闭及离开窗口的命令

命令	说明
:quit[!] ^Wq ^W^Q	离开当前窗口
:close[!] ^Wc	关闭当前窗口
:only[!] ^Wo ^W^O	让当前的窗口成为唯一的窗口

摘要

各位现在应该感受到了Vim提供了许多与窗口相关的功能，提升了编辑工作的能力。Vim可让我们轻松且即时地创建并删除窗口。除此之外，还提供了缓冲区命令。缓冲区作为底层文件管理基础结构，Vim可据此管理窗口编辑。这一点再一次证明，Vim如何既为初学者带来多窗口编辑工具，同时又为有经验的用户提供针对其窗口经验所需的工具。

第十二章

Vim脚本

有时候，只有自定义行为还不足以应付你的编辑环境。Vim已让我们可在.vimrc文件里定义所有首选设置（favorite setting），但各位或许还想要更动态或更“即时”的配置。Vim脚本（script）可以实现你的需求。

从查看缓冲区内容到处理意外的外部因素，Vim的脚本语言能让我们完成复杂的任务并根据个人需求作决定。

如果你有Vim配置文件（.vimrc、.gvimrc或两者均有），其实你就已经在编写 Vim里的脚本，只是自己不知道罢了。所有Vim命令与选项都是脚本的有效输入。Vim也提供所有在其他语言中常见的标准流控制（例如if...then...else、while）、变量、函数……

本章中，我们将探索一个范例并逐步编写一份脚本。我们会讨论简单的结构，试着利用Vim内置函数，并检查必须考虑的准则以设计出行为良好、可以预测的Vim脚本。

你最爱什么色调？

让我们从最简单的配置开始——自定义首选的环境配色方案。这项工作既简单，而且使用了Vim脚本的基础之一——单纯的Vim命令。

Vim有17种自定义配色方案。在.vimrc或.gvimrc文件里加上colorscheme命令，即可选择并打开某种配色方案。作者群之一最喜欢的“低调”配色方案是“沙漠”色调：

```
colorscheme desert
```

把类似上例的colorscheme加入配置文件后，每次使用Vim编辑时，都会看到你最喜欢的配色。

这个脚本初体验只是小菜一碟。如果你对配色方案的选择比较复杂怎么办？如果你喜欢使用多种色系的配色方案呢？如果时间与首选设置有关系呢？Vim脚本均能让我们轻易地办到。

注意： 根据时间早晚而选择不同的配色方案听起来有点老套，但即使是Google，也会在一天里根据时间而改变*iGoogle*主页的配色。

条件执行

本书作者群中，有人喜欢把一天分成四个时段，每个时段都有专用的配色方案：

`darkblue`
: 午夜至早上六点。

`morning`
: 早上六点至中午。

`shine`
: 中午至晚上六点。

`evening`
: 晚上六点至午夜。

我们将为上述时段构建一个嵌套的`if...then...else...`代码块。在这个块中，有几种不同的语法可供使用。其中一种较为传统，具有很明显的语法编排特征：

```
if cond expr
  一行Vim代码
  另一行Vim代码
  ...
elseif some secondary cond expr
  本案例所用的代码
else
   若未符合上述任何案例时所用的代码
endif
```

`elseif`与`else`块可供选择是否使用，而且你可以加入多个`elseif`块。Vim也容许更为简洁、类似C的代码结构：

```
cond ? expr 1 : expr 2
```

Vim检查条件句*cond*。如果条件句成立，则执行*expr 1*；反之，则执行*expr 2*。

使用strftime()函数

知道如何有条件地执行代码后，我们需要知道时段。Vim有个返回时间信息的内置函数。以我们的需求而言，应使用strftime()函数。strftime()接受两个参数，第一个参数定义时间字符串的输出格式（格式因系统而异，而且不能跨平台使用，所以在选择格式时务必小心。幸好，大多数常用格式在各个系统中都算常见）。第二个是可选参数，是从1970年1月1日开始计算的秒数（标准的C时间表示方式）。本可选参数的默认值为当前时刻。以我们的范例而言，可以使用时间格式%H，形成strftime("%H")strftime("%H")。因为只需要小时的信息，即可决定采用的配色方案。

知道如何设置条件代码后，还必须使用Vim内置函数取得时间信息，以选择符合要求的配色方案。把下列代码放到你的.vimrc文件里：

```
" progressively check higher values... falls out on first "true"
" (note addition of zero ... this guarantees return from function is numeric
if strftime("%H") < 6 + 0
colorscheme darkblue
echo "setting colorscheme to darkblue"
elseif strftime("%H") < 12 + 0
colorscheme morning
echo "setting colorscheme to morning"
elseif strftime("%H") < 18 + 0
colorscheme shine
echo "setting colorscheme to shine"
else
colorscheme evening
echo "setting colorscheme to evening"
endif
```

请注意，我们引入了另一个Vim脚本命令，echo。我们加入echo以便反映当前的配色方案；它也能让我们检验代码是否确实执行了，是否产生了想要的结果。echo的消息应该显示在Vim的命令行状态窗口，或以对话框的形式出现，根据它在启动序列中的位置而定。

注意： 当我们发出colorscheme命令时，使用的配色方案的名称（例如desert）不需加上引号；但在发出echo命令时，则需加上引号（"desert"），这项差别很重要！

以上例的colorscheme命令为例，我们发出一个直接的Vim命令，这个命令的参数为字面量（literal）。如果我们加入引号，引号将被colorscheme解释为配色方案名称的一部分，将产生错误，因为所有配色方案名称里都没有引号。

但是，echo则把没有引号的词汇当成表达式（一种有返回值的运算工具）或函数。因此，我们需要把选择的配色方案名称加上引号。

变量

如果你是个程序员，大概已经从刚才的脚本里发现了问题。虽然对我们的任务不会是太大的问题，但上例在每个判断条件的地方都调用strftime()函数以检查时间。从技术上而言，我们有条件地在检查同一件事，只是以表达式的形式估算多次，有可能在执行途中就遇到改变条件判断结果的状况。

与其每次都执行这个函数，不如只对函数估算一次并把结果存储在 Vim脚本变量里。而后既可于条件句中随意使用变量，又不需付经常执行函数调用所需的成本。

Vim的变量非常简单，但有几件需要知道与管理的事项，尤其是必须管理我们的变量作用域（scope）。Vim有一套定义变量作用域的惯例，依赖于变量名称的前缀。前缀包括：

`b:`
: 在单一Vim缓冲区里被辨识的变量

`w:`
: 在单一Vim窗口里被辨识的变量

`t:`
: 在单一Vim分页里被辨识的变量

`g:`
: 全局变量——也就是能在任何地方被辨识

`l:`
: 在函数内被辨识的变量（局部变量）

`s:`
: 在来源的Vim脚本里被辨识的变量

`a:`
: 函数的参数

`v:`
: Vim变量——由Vim控制（也是全局变量）

注意： 如果不以前缀定义Vim变量的作用域，当变量定义在函数外时，其默认为全局变量（:g）；定义在函数内时，其默认为局部变量（:l）。

可使用let命令指派变量值：

```
:let var = "value"
```

根据目的，可以随需求（包括上下文）定义变量，因为我们只会使用一次（但这点稍后会改变）。目前先不使用前缀，则Vim把我们的变量视为默认的全局变量。变量取名为currentHour。只需把strftime()的结果指派给这个变量一次，就得到更有效率的脚本了：

```
" progressively check higher values... falls out on first "true"
" (note addition of zero ... this guarantees return from function is numeric)
let currentHour = strftime ("%H")
echo "currentHour is " currentHour
if currentHour < 6 + 0
  colorscheme darkblue
  echo "setting colorscheme to darkblue"
elseif currentHour < 12 + 0
  colorscheme morning
  echo "setting colorscheme to morning"
elseif currentHour < 18 + 0
  colorscheme shine
  echo "setting colorscheme to shine"
else
  colorscheme evening
  echo "setting colorscheme to evening"
endif
```

引入新变量colorScheme，我们可以再减少一点代码。这个变量保留根据时间决定的配色方案值，它的名称中采用大写字母“S”与命令colorscheme区隔。但其实可以使用大小写完全相同的字母当变量名称。Vim能根据上下文而判断出它是命令还是变量。

注意：请注意点号（.）表示法与echo命令的搭配使用。这个运算符串联表达式与字符串形成echo最后显示的内容。本例中，我们串联字符串“setting color scheme to”与指派给colorScheme变量的值。

```
" progressively check higher values... falls out on first "true"
" (note addition of zero ... this guarantees return from function is numeric
let currentHour = strftime("%H")
echo "currentHour is " . currentHour
if currentHour < 6 + 0
  let colorScheme ="darkblue"
elseif currentHour < 12 + 0
  let colorScheme = "morning"
elseif currentHour < 18 + 0
  let colorScheme = "shine"
else
  let colorScheme = "evening"
endif
echo "setting color scheme to" . colorScheme
colorscheme colorScheme
```

警告： 在上例的脚本中，我们对于执行命令做了错误假设。如果各位照着上例编写，想必已经知道错误为何。我们将在下一节更正这项错误。

execute命令

到目前为止，我们改进了选择配色方案的方式，但上一次改变有点走错路。我们的原意是根据时间直接执行配色方案。上一次改进看起来正确，但在定义了保留配色方案值的变量（colorScheme）后，下述命令：

```
colorscheme colorScheme
```

返回了如图12-1所示的错误消息。

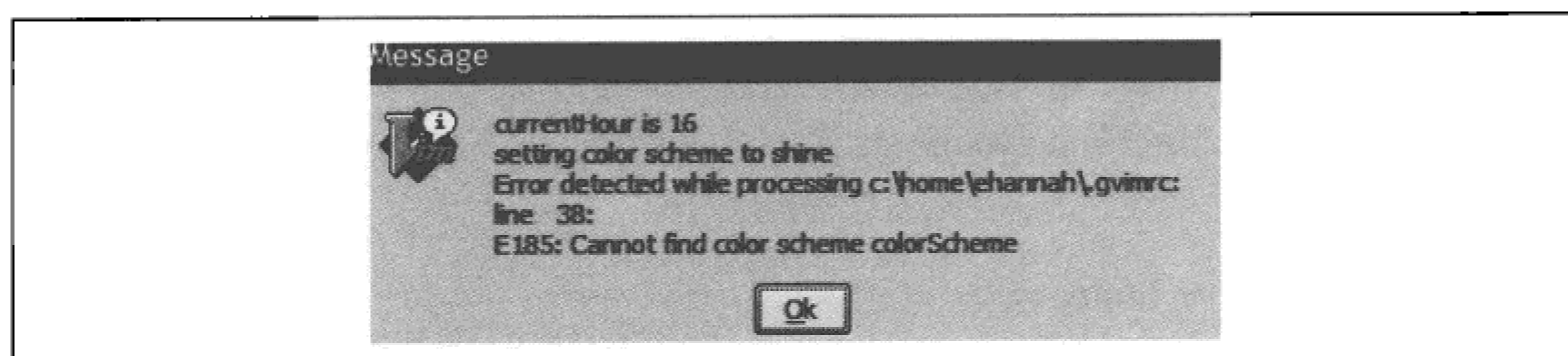


图12-1：colorscheme colorScheme 错误消息

我们需要一个执行Vim命令且指向参数而非指向字面字符串（如darkblue）的方式。为此，Vim提供了execute命令。传给它一个命令时，它会估算变量与表达式，并把结果代入命令的适当位置。我们可以利用这项功能，配合使用前一节讲到的串联，即可传递变量值给colorscheme命令：

```
execute "colorscheme " . colorScheme
```

本处使用的语法（尤其是引号）可能让人看了很理不出头绪。execute预期接收变量或表达式的值，但colorscheme只是一个字符串。我们不希望execute实际估算colorscheme的值，只是希望它接受这个名称。所我们用引号括起名称，把它转变为字面字符串。同时，我们在结尾引号前加上一个空格，这一点很重要，因为在命令与值中间需要有一个空格。

变量colorScheme必须在引号外才会被execute估算。execute的行为可以这么理解：

- 估算单纯的词（不加引号）时，词将被视为变量或表达式，execute将把传入部分替换为估算的结果。
- 引号括起的字符串被视为文字，execute不会试着估算字符串以及返回一个值。

使用execute可修正我们的错误，Vim现在如预期般地载入指定配色方案了。

在进入 Vim 后，即可确认是否载入了适当的配色方案。可用colorscheme命令设置它自身的变量colors_name。除了回显设置在脚本中的变量值，我们也可以手动执行echo命令并检查变量 colors_name，以确认脚本是否根据时间而执行了正确的colorscheme命令：

```
echo colors_name
```

定义函数

我们已经创建了一个运作良好的脚本。现在让我们创建可以在会话中任何时刻执行的代码，而不只是在Vim启动时执行。我们很快就会提供一个范例，但首先需要创建包含脚本代码的函数。

Vim 使用function...endfunction语句让我们定义函数。以下是用户自定义函数的样本结构：

```
function myFunction (arg1, arg2...)
  line of code
  another line of code
endfunction
```

我们的代码可以轻易地转变为函数。请注意，我们不需要传入任何参数，所以函数定义中的括号里为空白：

```
function SetTimeOfDayColors()
  " progressively check higher values... falls out on first "true"
  " (note addition of zero ... this guarantees return from function is numeric)
  let currentHour = strftime("%H")
  echo "currentHour is " . currentHour
  if currentHour < 6 + 0
    let colorScheme = "darkblue"
  elseif currentHour < 12 + 0
    let colorScheme = "morning"
  elseif currentHour < 18 + 0
    let colorScheme = "shine"
  else
    let colorScheme = "evening"
  endif
  echo "setting color scheme to" . colorScheme
  execute "colorscheme " . colorScheme
endfunction
```

注意： Vim的用户自定义函数名称必须以大写字母开头。

现在.gvimrc文件里定义了一个函数。如果我们不调用函数，其中的代码也不会执行。请使用Vim的call语句调用函数。以我们上面创建的函数为例：

```
call SetTimeOfDayColors()
```

现在，我们随时可于Vim的一个会话中设置配色方案了。方法之一是只把上述的call语句行放入.gvimrc，其结果与稍早的范例相同，差别只在于原本的范例没有使用函数。但在下一节，我们会看到另一种Vim技巧，可重复调用函数，自动于会话中设置配色方案，因此配色方案会动态改变！当然，如此一来又引入了其他需要面对的问题。

不错的 Vim取巧诀窍

前一节，我们定义了一个Vim函数SetTimeOfDayColors()，并调用了一次用于定义配色方案。如果我们想重复检查时间，并根据时间改变配色方案呢？显然，只在.gvimrc里调用一次不能实现这项需求。为了解决问题，我们引入一项很棒的Vim技巧——使用statusline选项。

大多数Vim用户把Vim的状态行视为理所当然的存在。默认情况下，statusline没有值，但可把状态行定义为显示Vim可取得的（几乎）任何信息。而且因为状态行能显示动态消息（例如当前的行数与列数），Vim在任何状态改变时均重新计算并显示statusline的值，即Vim的任何动作几乎都能触发statusline的重绘动作。我们将利用这项特点调用配色方案设置函数并动态地改变配色方案。但很快我们也将看到这个技术的缺点。

statusline能接受表达式并在计算后于状态行显示结果。也可以引入函数。我们利用这项特点，于每次更新状态行时调用SetTimeOfDayColors()。更新的次数将很频繁，因为这项功能覆盖所有默认的状态行，而我们不想失去默认取得的重要信息，因此可在下列的状态行初始定义中合并丰富的信息：

```
set statusline=%<%t%h%m%r\ \ %a\ %{strftime(\"%c\")}%=0x%B\
    \\ line:%l,\ \ col:%c%V\ %P
```

注意： statusline的定义分成两行。只要第一个非空字符是反斜线，Vim就会把这一行视为前一行的延续。如果各位使用我们的定义，请务必精确地复制与输入。如果不能运算，你可以再使用未定义的statusline重新开始。

关于字符前加上百分号（%）后的意义，请参考Vim的说明文档。上述定义产生如下状态行：

```
ch12.xml     Wed 13 Feb 2008 06:24:25 PM EST        0x3C line:1,    col:1 Top
```

本节的重点并非显示状态行，而是在计算函数时利用statusline选项。

现在把SetTimeOfDayColors()函数加入statusline。需使用+=取代一般等号（=），才能于末端添加新内容，而非取代稍早定义的内容：

```
set statusline += \ %{SetTimeOfDayColors()}
```

我们的函数现在成为状态行的一部分了，虽然它不会提供状态行的有趣信息，但会检查时间，并随着时间流逝而于适当的时候更新配色方案。各位看出这里的问题了吗？我们现在写了一个每次Vim的状态行更新就检查时间的Vim脚本函数。在稍早的章节中，我们努力减少对strftime()的调用以增进效率，而现在却于会话里增加了次数多到数不清的函数调用。

当我们在恰当的时间估算statusline，它会产生我们想要的效果——改变配色方案。但如同我们的定义，它会检查时间并重设配色方案，不管是否需要改变。在下一节中，我们的检查会更有效率，这就是在函数外使用全局变量。

以全局变量转变Vim脚本

前一次对Vim脚本调整后，我们几乎达到了预期的行为。每次更新Vim的状态行都会调用我们的函数，但因为发生的次数太过频繁，故数个层面均可能出现问题。

首先，因为它太频繁地被调用，可能会对计算机处理器造成负担。幸好，对今天的计算机而言，这点已经不太构成问题。但如此频繁地反复地重新定义配色方案，仍是种不好的形式。如果这是唯一的问题，或许这个脚本可以姑且视为已完成，不必再进一步微调。然而，并非如此。

如果各位随着我们的步骤编写脚本，应该已经知道问题何在了——在我们于编辑会话中行动时，会经常性地发生配色方案重建，造成伤眼又烦人的闪烁，因为每次配色方案的定义（即使与当前配色方案相同），都需要Vim重新读入定义配色方案的脚本，重新解读文字，重新运用所有语法高亮显示的规则。即使是具备极强大计算能力的计算机，也不太可能提供足够的图形处理能力以应付经常性的更新而不会闪烁。我们需要处理这个问题。

我们可以先定义配色方案，然后每次在条件块中判断配色方案是否需改变、是否需执行后续的重定义与重绘画面。实现的方式是利用colorscheme命令设置全局变量colors_name。让我们重定义函数，考虑使用全局变量：

```
function SetTimeOfDayColors()
  " progressively check higher values... falls out on first "true"
  " (note addition of zero ... this guarantees return from function is numeric)
```

```
  let currentHour = strftime("%H")
    if currentHour < 6 + 0
    let colorScheme = "darkblue"
  elseif currentHour < 12 + 0
    let colorScheme = "morning"
  elseif currentHour < 18 + 0
    let colorScheme = "shine"
  else
    et colorScheme = "evening"
  endif

  " if our calculated value is different, call the colorscheme command.
  if g:colors_name !~ colorScheme
    echo "setting color scheme to " . colorScheme
    execute "colorscheme " . colorScheme
  endif
endfunction
```

上例似乎解决了我们的问题，但却出现另一个问题，如图12-2所示。

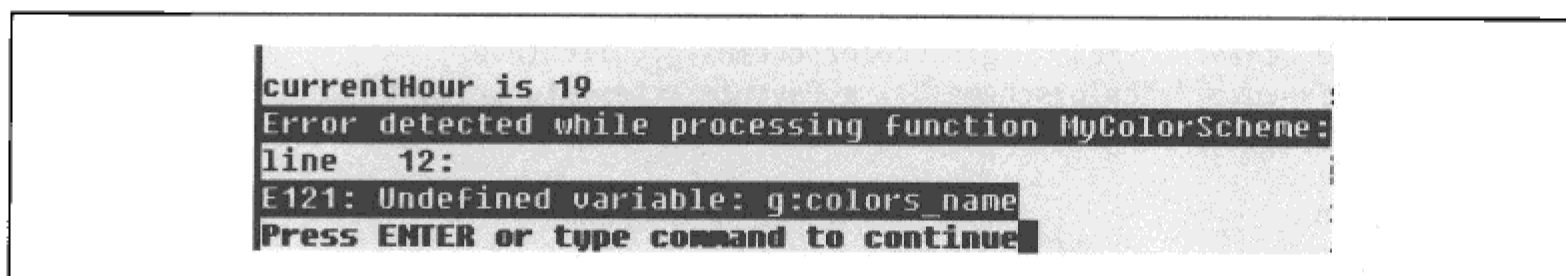


图12-2：未定义的变量

看起来Vim在指向未定义的变量时采取了严格的态度，但colors_name为什么也会遇到这个问题？我们知道colorscheme设置了这个变量，甚至小心地使用了前缀g:指出它是个全局变量。但是这个函数首次执行时，g:colors_name中没有值，也尚未定义，因为colorscheme命令尚未执行。只有在这个命令执行后，才能安全地检查到g:colors_name。

这一点很容易修正，而且有两种修正方式。在你的.gvimrc文件里输入下列定义之一：

```
let g:colors_name = "xyzzy"
```

或

```
colorscheme default
```

两条语句均可于开始会话时即刻定义全局变量，从而让函数中的比较一直都有效。现在我们有了一个动态而且有效率的函数。下一节将进行最后一项改进。

数组

如果能够只提取配色方案值而不用增加if...then...else块就更完美了。利用Vim的数组，可以改进脚本并大幅增加可读性。

把变量值定义为方括号（[...]）中由逗号分隔的一串值，即可定义Vim数组。我们为函数引入一个命名为Favcolorschemes的数组。此数组的定义可在函数作用域中进行，但考虑到在会话的其他部分访问数组的可能性，我们在函数外将其定义成全局数组：

```
let g:Favcolorschemes = ["darkblue", "morning", "shine", "evening"]
```

这一行应该放入你的.gvimrc文件。现在我们可以用它的下标（subscript）引用任何在数组变量g:Favcolorschemes中的值，从元素0开始。例如g:Favcolorschemes[2]等于字符串"shine"。

利用Vim处理数学函数的方式（整数除法的结果仍然是整数，余数则被砍掉），可以快速而轻易地根据时间取得首选配色方案。让我们看看最终版的自定义函数：

```
function SetTimeOfDayColors()
  " currentHour will be 0, 1, 2, or 3
  let g:CurrentHour = (strftime("%H") + 0) / 6
  if g:colors_name !~ g:Favcolorschemes[g:CurrentHour]
    execute "colorscheme " . g:Favcolorschemes[g:CurrentHour]
    echo "execute " "colorscheme " . g:Favcolorschemes[g:CurrentHour]
    redraw
  endif
endfunction
```

恭喜！你刚刚创建了一个完整的Vim脚本，其中考虑到了许多创建任何好的脚本时可能需要的因素。

通过脚本动态配置文件类型

让我们再看一个很不错的脚本范例。一般而言，编辑新文件时，Vim用于判断并设置filetype的唯一线索就是文件的扩展名（extension）。例如，.c表示该文件为C代码。Vim能轻易辨别这项线索并载入正确的行为，从而让编辑C程序更为轻松。

但并非所有文件都需要扩展名。就像shell脚本采取扩展名.sh虽然已成为惯例，但有人不喜欢使用，还有人是在创建数千份脚本后才知道有这个惯例。事实上Vim具备良好的训练，只需判读文件内容，不需扩展名的支持也可以辨识出shell脚本。然而，这个方式只能用在文件提供了一些内容以供判断文件类型（如第二次编辑）后。修改Vim脚本能弥补这个遗憾！

自动命令

在我们的第一个脚本范例里，依赖Vim更新状态行的习性，把函数“隐藏”在状态行里，以根据时间设置配色方案。而要编写能判断文件类型的脚本，则需依赖更为正式的Vim习惯——自动命令（autocommand）。

自动命令包括任何合格的Vim命令。Vim使用事件执行命令。Vim事件包括：

`BufNewFile`
: 在 Vim 开始编辑新文件时触发相关联的命令

`BufReadPre`
: 在 Vim 移向新缓冲区前触发相关联的命令

`BufRead, BufReadPost`
: 在编辑新文件时触发相关联的命令，但需在读入文件后

`BufWrite, BufWritePre`
: 把缓冲区的内容写入文件前触发相关联的命令

`FileType`
: 在设置filetype后触发相关联的命令

`VimResized`
: 在改变Vim窗口尺寸后触发相关联的命令

`WinEnter, WinLeave`
: 分别在进入、离开Vim窗口时触发相关联的命令

`CursorMoved, CursorMovedI`
: 分别在每次光标进入正常（normal）模式、插入（insert）模式时触发相关联的命令

总共有80个Vim事件。任何一个事件都可定义于事件发生时执行的`autocmd`中，其格式如下：

```
autocmd [group] event pattern [nested] command
```

上述格式中：

group
: 可选的命令组（稍后说明）

event
: 触发命令的事件

pattern
: 匹配文件名的模式，用于找出应执行命令的文件

nested
: 如果出现，表示这个自动命令能放在其他自动命令中

command

当事件发生时执行的Vim命令、函数或用户自定义的脚本

对我们的范例来说，我们的目的在于辨识任何新打开文件的类型，所以*pattern*为*。

再来决定触发脚本的事件。因为想要尽早辨识出文件类型，跳出两项候选事件：CursorMovedI与CursorMoved。

CursorMoved于光标移动时触发事件，使用它似乎有点浪费，因为只是移动光标，不太可能提供关于文件类型的更多信息。CursorMovedI是于输入文本时触发，似乎是个不错的选择。

我们必须设计负责该工作的函数，称之为CheckFileType。现在已有足够信息可定义autocmd，如下所示：

```
autocmd CursorMovedI * call CheckFileType()
```

检查选项

在我们的CheckFileType函数中，需要查看filetype选项的值。Vim脚本利用特定变量获取选项值，方法是在选项名称前（本例为filetype）加入前缀（&）字符。因此我们将在函数中使用变量&filetype。

先从简易版的CheckFileType函数开始：

```
function CheckFileType()
  if &filetype == ""
    filetype detect
  endif
endfunction
```

Vim命令filetype detect是个放置在$VIMRUNTIME目录下的Vim脚本。它检查许多标准，试着为文件指派一个类型。通常这个命令只需运行一次。如果文件是新文件，而filetype不能判断文件类型时，编辑会话不能指派语法格式。

有个问题：每次光标在插入模式中移动时，都会调用我们的函数以检测文件类型。应首先确认是否已经有文件类型，可能表示我们的函数在上一次已经成功执行，因此不需要再运行一次。此处不特别考虑异常状况，例如辨识错误，或是遇到原本以A语言设计后来却更改为B语言的程序。

先编辑一份新的shell脚本并看看它的结果：

```
$ vim ScriptWithoutSuffix
```

输入下列内容：

```
#! /bin/sh

inputFile="DailyReceipts"
```

现在，Vim打开了语法颜色高亮显示，如图12-3所示。

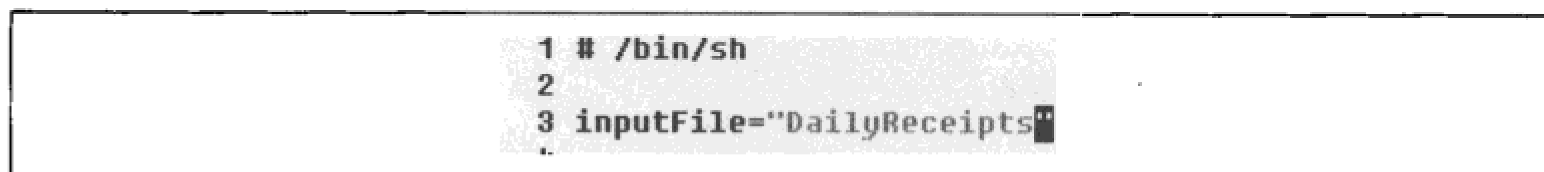


图12-3：检测到新文件的文件类型

从图中可看到，Vim把字符串标为灰色，但黑白印刷呈现不出# /bin/sh标置为蓝色，inputFile是黑色，"DailyReceipts"则为紫色的效果。而且，这不是shell语法标准的配色。通过命令set filetype，发现filetype选项显示如图12-4的消息。



图12-4：检测到conf文件类型

Vim指派文件类型为conf，但并非我们想要的结果。哪里出了问题？如果你尝试过我们的范例，就会看到Vim在输入第一个字符#后立即指派文件类型，也就是第一个CursorMovedI事件发生时。Unix实用程序与daemon均以#字符开始注释，所以Vim假设一行的开头字符为#时，表示它是配置文件的注释。我们必须教Vim耐心地多等一下再判断。

让我们把函数修改为允许更多上下文纳入判断考虑中，而不是第一个判断标准出现时就忙着检测文件类型，例如允许用户输入20个字符后再检测。

缓冲区变量

我们需要对函数引入变量，告诉Vim先别试着检测文件类型，直到CursorMovedI自动命令调用函数的次数超过20次时。判断是否为新文件以及我们想输入文件里的字符数量，都需引用同一个缓冲区。换句话说，编辑会话的其他缓冲区中的光标移动应该都不计入调用函数的次数里。因此，我们使用缓冲区变量b:countCheck。

接下来，我们修正函数以检查光标是否已在插入模式中移动超过20次（即大约输入 20个字符），并检查文件类型是否已被设置：

```
function CheckFileType()

let b:countCheck += 1

" Don't start detecting until approx. 20 chars.
 if &filetype == "" && b:countCheck > 20
   filetype detect
 endif
endfunction
```

出现如图12-5所示的错误消息。

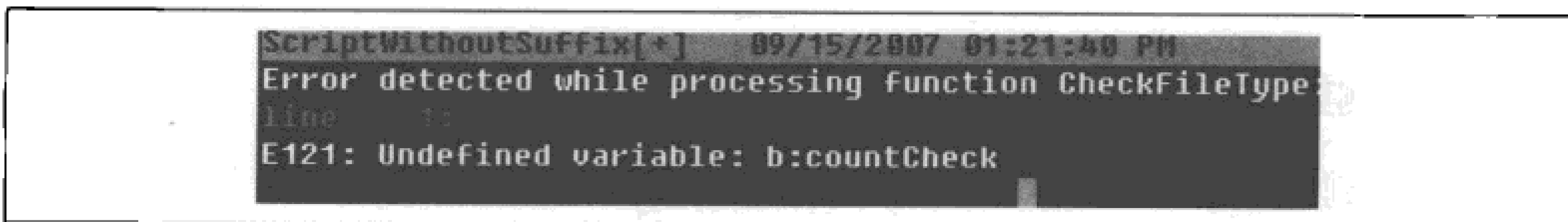


图12-5：b:countCheck 产生一个“未定义的”错误

这是个很眼熟的错误消息，稍早才发生过，我们又遇到了在变量定义前就检查变量的问题。这一次是我们的错，因为我们的脚本应该负责定义b:countCheck。下一节将详细说明这一点。

exists()函数

知道如何管理自己的所有变量与函数是很重要的。Vim需要我们定义每一个变量与函数，所以在任何运算引用到变量或函数前，它们必须已经存在。借由检查b:countCheck是否存在并用之前所述的:let命令指派一个值给它，即可轻易地解决范例脚本中的错误：

```
function CheckFileType()

 if exists("b:countCheck") == 0
  let b:countCheck = 0
endif

let b:countCheck += 1

 " Don't start detecting until approx. 20 chars.
 if &filetype == "" && b:countCheck > 20
   filetype detect
 endif
endfunction
```

现在再测试一下代码。图12-6展示了达到20个字符限制前的状态，图12-7则是输入第21个字符时的效果。

```
1 #! /bin/sh
2
3 inputFil
~
```

图12-6：还没有检测到文件类型

```
1 #! /bin/sh
2
3 inputFile
~
```

图12-7：检测到文件类型

/bin/sh突然加上了语法颜色高亮显示。使用set filetype快速检查一下，确认Vim已经正确指定了文件类型，如图12-8所示。

```
ScriptWithoutSuffix[+]
  filetype=sh
```

图12-8：正确检测

基于所有实际用途，这已是一个完整且令人满意的方案。但为了让格式更好看，再加上另一项检查以阻止Vim在输入200个字符后还试着检测文件类型：

```
function CheckFileType()

if exists("b:countCheck") == 0
  let b:countCheck = 0
endif

let b:countCheck += 1

 " Don't start detecting until approx. 20 chars.
 if &filetype == "" && b:countCheck > 20 && b:countCheck < 200
   filetype detect
endif
endfunction
```

现在，每次在光标移动时Vim都会调用我们的CheckFileType函数，会出现一点经常性成本，因为初始检测在确定函数类型或达到200个字符的阈值后才会跳出函数。虽然这样大概已是合理运作且兼顾最少量处理的整体成本，但我们将继续讨论更多机制，带来更完善、更令人满意的解决方案，不只能使整体成本降至最低，也确实地在我们不需要这个函数时“挥一挥衣袖，不带走一片云彩”。

注意： 各位可能发现，我们对于20个字符的阈值计算的确切定义有点模糊。这是刻意的模糊不清。因为计算光标移动时，在插入模式中，假设每次光标移动都对应到一个新字符，加入"足够的"上下文文本，以利于CheckFileType()判断文件类型，这是项合理的假设。然而，这样在插入模式中的所有光标移动都会被计算，所以任何退格以更正错字的移动也计算在内。要确认这一点，可在我们的范例中键入#，然后按退格键；再键入一次#，按退格键；重复10次，第11次就会出现标上颜色的#，文件类型则（不正确地）被设置为conf。

自动命令与组

到目前为止，我们的脚本忽略了每次移动光标就调用函数的副作用。虽然可借由合理检查避免不必要地调用繁复的filetype detect命令，以降低整体成本，但如果函数的最低成本也很昂贵呢？我们需要一个在不需要代码时停止调用的方式。因此，我们借用Vim对自动命令组的表示法，同时借用它们根据组关联删除命令的能力。

范例的修改首先是通过CursorMovedI事件为函数调用与一个组制造关联。Vim提供了augroup命令，它的语法是：

```
augroup groupname
```

这时所有后续的autocmd定义均与*groupname*组产生关联，直到出现如下语句：

```
augroup END
```

（还有一个命令的默认组，不在augroup块里输入）。

现在为前面的autocmd定义与我们的组制造关联：

```
augroup newFileDetection
autocmd CursorMovedI * call CheckFileType()
augroup END
```

前例中以CursorMovedI触发的函数现在成为自动命令组newFileDetection的一部分。下一节将全面探讨这项功能。

删除自动命令

为了尽可能有效率地实现我们的函数，我们努力让它只在需要时维持效用。我们希望在函数完成功能后，即解除对它的引用（也就是检测到文件类型或确定检测不出类型的时候）。Vim可通过引用事件而删除自动命令，删除对象需匹配文件名称的模式或属于某个组。

```
autocmd! [group] [event] [pattern]
```

通常在Vim中用于“强制”的字符——感叹号（!）接在关键字autocmd后，表示与组、事件或模式相关联的命令将被删除。

因为我们稍早已让函数与用户自定义组newFileDetection产生关联，所以可通过组控制函数，并在自动命令的删除语法中引用组以删除函数。如下所示：

```
autocmd! newFileDetection
```

即可删除所有与newFileDetection关联的自动命令，此例为删除我们的函数。

确认自动命令的定义与删除，均于启动时（创建新文件时）使用如下命令查询Vim：

```
autocmd newFileDetection
```

Vim的响应如图12-9所示。

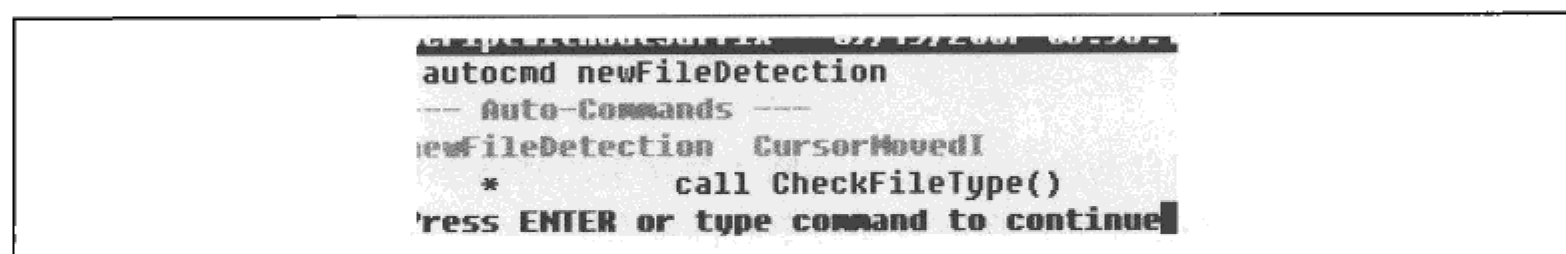


图12-9：autocmd newFlieDetection命令的响应

在检测出文件类型并予以指派或达到200个字符的阈值后，我们不再需要自动命令的定义。所以加上对代码的最后一项修改，结合我们对augroup的定义、autocmd命令以及自定义函数后，.vimrc的内容如下所示：

```
augroup newFileDetection
autocmd CursorMovedI * call CheckFileType()
augroup END

function CheckFileType()

  if exists("b:countCheck") == 0
    let b:countCheck = 0
  endif

  let b:countCheck += 1

  " Don't start detecting until approx. 20 chars.
  if &filetype == "" && b:countCheck > 20 && b:countCheck < 200
    filetype detect
  " If we've exceeded the count threshold (200), OR a filetype has been detected
  " delete the autocmd!
  elseif b:countCheck >= 200 || &filetype != ""
    autocmd! newFileDetection
  endif
endfunction
```

开始出现语法颜色高亮显示后，我们可以输入与缓冲区输入相同的命令，以确认函数是否会自我删除：

```
autocmd newFileDetection
```

Vim的响应如图2-10所示。

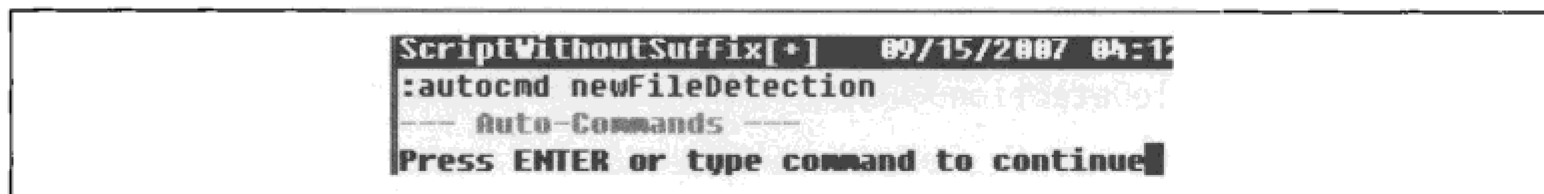


图12-10：自动命令组的删除标准达到之后

请注意，现在的newFileDetection组中没有定义自动命令了。下列命令可删除自动组（auto group）：

```
augroup! groupname
```

但并不会删除已关联的自动命令，而且Vim将于每次引用到这些自动命令时创建错误状态。因此，请确定已删除组中的所有自动命令后再删除组。

警告： 不要把删除自动组（auto group）误以为删除自动命令。

恭喜！你已经完成第二份Vim脚本。这次的脚本扩充你对于Vim的知识，也使你窥见从脚本编码可取用的各种不同功能。

关于Vim脚本编码的其他思考

偌大的Vim脚本编码世界里，我们只讨论了一小部分，希望大家都能体会到Vim的威力。在脚本中几乎能编码一切使用Vim交互的行为。

在本节中，我们将讨论一个包含在Vim内置说明文档中的优秀范例，深入了解了稍早提过的概念，并仔细研究几种功能。

一个好用的Vim脚本范例

在Vim的内置说明文档中包含一个好用的脚本，我们觉得各位可能用得到。这份脚本专门处理在 HTML 文件的meta行中保持当前时间戳（采计算机纪元格式）的问题，但其也可以轻易用于许多其他文件类型，只要于文件的文本中保存最新的文件修改时间，其就有所助益。

先看一个基本完整的范例（我们稍微做了点调整）：

```
autocmd BufWritePre,FileWritePre *.html   mark s|call LastMod()|'s
fun LastMod()
  " if there are more than 20 lines, set our max to 20, otherwise, scan
  " entire file.
  if line("$") > 20
    let lastModifiedline = 20
  else
    let lastModifiedline = line("$")
  endif
  exe "1," . lastModifiedline . "g/Last modified: /s/Last modified:
      .*/Last modified: " .
  \ strftime("%Y %b %d")
endfun
```

简要分析一下autocmd命令：

`BufWritePre, FileWritePre`
: 这是触发命令的地方。在本例中，Vim在文件或缓冲区写入存储设备前执行自动命令。

`*.html`
: 对任何文件名结尾为.html的文件执行自动命令。

`mark s`
: 改变此项以从原文件中做准备。此处不采用ks，而是使用效果相等但意义更为明显的mark s命令。它会在文件中创建一个标记位置，命名为s，以备日后返回到这个位置。

`|`
: 管道字符，用于分隔在自动命令定义中执行的许多Vim命令。此处只是单纯用于分隔，与UNIX shell的相同字符没有关系。

`call LastMod()`
: 调用用户定义的LastMod函数。

`|`
: 参考前述说明。

`'s`
: 回到标记为s的位置。

这份脚本值得核对一下，请编辑一份.html文件，在其中加入“Last modified: ""”，并发出w命令。

注意： 这个范例的确很有用，但对于它的目标——取代 HTML的meta语句而言，不够正确。更适当地说，如果它真的想针对meta语句，替换时应该寻找meta语句中的content=...部分。不过，这个范例仍然是解决这个问题的好的开始，而且对解决其他文件类型的相同问题而言也是好例子。

再谈变量

现在进一步讨论Vim变量的构成以及使用方式。Vim有5种变量类型：

数字（number）
: 有符号的32位数字。数字能以十进制、十六进制（如0xffff）、八进制（如0177）形式表示。

字符串（string）
: 一串字符。

函数引用（funcref）
: 对函数的引用。

列表（list）
: Vim版的数组。它是有序的值“列表”，可包含任何Vim值的混合作为它的元素。

字典（dictionary）
: Vim 版的散列（hash），通常也称为关联数组。它是许多对数组值的无序集合，位于前面的一个值被当成键(key)，用于取得关联值（value）。

Vim在上下文允许时执行便利的变量转换，大多数是明显的字符串与数字的转换。为了安全起见，如同第一个脚本范例的做法，把字符串当成数字使用时，请确定转换时加上了0：

```
if strftime("%H") < 6 + 0
```

表达式

Vim以相当简单的方式估算表达式。表达式可以是很简单的数字或字面字符串，也可以复杂到是个复合语句，由表达式组成。

有一个重点值得注意：Vim的数学函数只能与整数一起运作。如果需要浮点数与精度，则需要一些扩展组件，例如由系统调用可以做数学运算的外部例程。

扩展组件

Vim提供不少扩展组件与其他脚本语言的接口。尤其值得注意的是，其中包括perl、

Python和Ruby，三种最受欢迎的脚本语言。请参考Vim的内置说明文档，以了解使用细节。

再讨论autocmd

在第208页的“通过脚本动态配置文件类型”一节中，范例使用autocmd命令，从调用自定义函数的地方输入事件。它非常有用，但别低估autocmd命令的简易用法。例如，你可以使用autocmd为不同文件类型调整特定Vim选项。

为不同的文件类型改变shiftwidth选项或许是个不错的范例。具有大量缩排与嵌套层次的文件类型或许能因较有节制的缩排而获益。例如定义HTML的shiftwidth为2，以免代码超出屏幕右侧，但定义C程序的shiftwidth为4。为了达成这项区别，请把如下代码行加入你的.vimrc或.gvimrc文件中：

```
autocmd BufRead,BufNewFile *.html set shiftwidth=2
autocmd BufRead,BufNewFile *.c,*.h set shiftwidth=4
```

内部函数

除了所有的Vim命令，我们还能使用大约200个内置函数。列举并说明所有函数已经超出了本书讨论范围，但知道可使用的函数种类也将很有帮助。下列分类取自Vim内置的帮助文件usr_41.txt：

字符串操纵

所有程序员期待的标准字符串函数都在这些函数里，范围从转换的例程到子字符串例程等等，难以尽数。

列表函数

一整个数组的数组函数。其贴切反映了perl中的数组函数。

字典（关联数组）函数

这些函数包括提取、操纵、验证与其他类型的函数。这些函数也很类似perl的散列函数。

变量函数

这些函数包括取得函数（getter）与设置函数（setter），用于在Vim窗口与缓冲区间移动变量。还有一个判断变量类型的type。

光标与位置函数

这些函数允许光标在文件与缓冲区间移动，并创建标记以便记忆与返回位置。另外还有提供位置信息的函数（例如光标所在的行与列）。

当前缓冲区中的文本的函数

这些函数操纵缓冲区的内部文本，例如改变行、获取行等等。还有搜索用的函数。

系统与文件操纵函数

包括在运行Vim的操作系统中导航的函数，例如可在路径中寻找文件、判断当前的工作目录、创建与删除文件等等。这一组还包括`system()`函数，用于为外部执行传递命令给操作系统。

日期与时间函数

这些函数能对日期与时间格式做各式各样的操纵。

缓冲区、窗口与参数列表函数

这些函数提供收集缓冲区信息及各个缓冲区参数信息的机制。

命令行函数

这些函数取得命令行位置、命令行，并设置命令行中的光标位置。

快速修正与位置列表函数

这些函数获取并修改快速修正列表（quick fix list）。

插入模式补全函数

这些函数用于命令及插入补全功能。

折叠函数

这些函数提供折叠（fold）用的信息，并展开关闭的折叠以显现文字。

语法与高亮显示函数

这些函数获取关于语法高亮显示组与语法ID的信息。

拼写函数

这些函数寻找可能的错误拼写并提供修正建议。

历史记录函数

这些函数取得、新增、删除历史记录。

交互函数

这些函数为用户提供行动（例如选择文件）的接口。

GUI函数

包含三个简易函数，分别可以取得当前的字体、GUI窗口的*x*坐标与GUI窗口的*y*坐标。

Vim服务器函数

这些函数与（可能的）远程Vim服务器通信。

窗口尺寸与位置函数

这些函数取得窗口信息并允许将窗口“所见”内容的保存与恢复。

其他函数

还有各式各样“其他”函数不能归类至前述分类中。其中有exists，检查Vim条目的存在与否；还有has，检查Vim是否支持某项功能。

资源

我们希望激起大家对Vim脚本的兴趣，也提供了开始接触Vim脚本的足够信息。关于Vim脚本，足以出一本专著来讨论。幸好，还有其他资源可以协助我们。

Vim本身就是很好的着手点，请参考专门说明脚本编码的网页*http://www.vim.org/ scripts/index.php*。在这里将找到超过2 000份可供下载的脚本。其所有内容均可阅览下载，也很欢迎各位参与脚本投票，甚至贡献自己的脚本。

在此也提醒大家，内置的Vim说明文档乃是无价之宝。我们推荐的最有收获的帮助主题是：

```
help   autocmd
help   scripts
help   variables
help   functions
help   usr_41.txt
```

也别忘记在Vim的运行时目录中也有大量Vim脚本。所有后缀是.vim的文件都是脚本，且它们对于根据范例学习编码，提供了绝佳的、有创意的测试园地。

动手玩一玩，这是最好的学习方式。



第十三章

图形化Vim（gvim）

关于vi及其同类品，长年以来一直因为缺少图形用户界面（GUI）而受到抱怨。尤其是置身Emacs与vi孰优孰劣的争论中时，vi缺少GUI这一点，是攻击它并非给初学者使用的王牌。

终于，vi同类品及相似产品总算创建了自己的GUI版本。图形化Vim称为gvim。就像其他vi同类品，gvim也提供可靠而且可扩展的GUI函数及功能。本章将说明其大部分常用功能。

有些gvim的图形化功能是为常用的Vim功能加上精美包装；有些则引入大多数计算机用户期待的点击功能。虽然熟练的Vim用户（包括本书的作者）可能一想到自己简朴的编辑器被接上图形用户界面就退避三舍，但gvim的构想与实现都很贴心。它提供了各层次用户所需的功能，缓和了Vim对新手而言太陡峭的学习曲线，并为专家级的用户带来额外的编辑能力，这是不错的妥协方案。

注意： MS Windows 版的gvim附有一个标示为“easy gvim”的菜单项。对于从未使用过Vim的用户而言，它的确非常有价值；但对于熟练的用户而言，它却一点都不简单。

本章首先讨论一般性的gvim图形用户界面概念与功能，有一节概述鼠标的交互。另外，对于不同gvim环境应该知道的差异与注意事项，我们也有更仔细的讨论，尤其是针对两大主流图形化平台——MS Windows与X Window System。对于其他平台也有简短描述，并为大家指引了可以了解更完整信息的资源。我们还提供了简短的GUI选项与概要表。

gvim概述

gvim具有Vim的所有功能、威力和特色，更加入了GUI环境下的便利与直观。从传统的

菜单到具有视觉强调效果的编辑帮助，gvim提供了时下用户期待的GUI体验。对于熟悉基于控制台的、文本环境的vi用户，gvim仍然提供了其熟悉的核心功能，也并未拖累为vi赢得强大编辑器声誉的典范。

启动gvim

当Vim编译加入GUI支持后，可直接发出gvim命令或在Vim命令后接-g选项来调用它。在Windows系统上，自我安装的可执行文件则会加入一项大家通常在安装后才不小心发现的有趣功能：它会新增一个Windows Explorer右键菜单项"Edit with Vim"（使用Vim编辑此文件）。借由整合至Windows环境，即可快速而简易地访问gvim。你可以拿一些以前不会考虑打开的文件做实验，尤其是二进制文件这类不寻常的文件。不过，编辑二进制文件具有潜在的危险性，提醒大家在编辑这类文件时千万提高警觉。

gvim能辨识的配置文件与选项与Vim用的稍有不同。gvim读入并执行两种启动文件：.vimrc，然后是.gvimrc。虽然可于.vimrc中定义gvim专用的选项，但最好定义在.gvimrc 里。如此可为Vim与gvim的自定义带来良好区别，也能确保启动时的适当行为。举例来说，**:set columns=100**在Vim中并不合格，将于启动Vim时产生错误。

如果系统有gvimrc（通常在$VIM/gvimrc目录下），则它会被自动执行。系统管理器可使用这个全系统的配置文件为用户设置通用的选项。如此就提供了基本配置，用户将可共享相通的编辑体验。

较有经验的Vim用户可加入自己喜欢的设置与功能。在gvim读入可选的系统配置后，还会在4个地方寻找额外的配置信息，寻找顺序如下，且若找到信息即停止寻找：

- 存储在$GVIMINIT环境变量里的exrc命令。
- 用户的gvimrc文件，通常存储在$HOME/.gvimrc。如果在此找到文件，则作为信息来源。
- 在Windows环境中，如果未设置$HOME，gvim则会在$VIM/_gvimrc中寻找（Windows 用户经常遇到这个情况，但对于安装了类似Unix系统的用户而言是项重大区别，后者很可能已经设置了$HOME变量，例如Unix工具里很受欢迎的Cygwin套件）。
- 如果_gvimrc在前三者中都未被发现，gvim最后则会寻找.gvimrc。

如果gvim找到可执行、有内容的文件，其文件将存储于$MYGVIMRC变量中，进一步的初始化则停止。

还有一个关于自定义的选项。如果，在如前所述的初始化层级顺序中，设置了exrc选

项：

```
set exrc
```

gvim将另外于当前的目录中寻找.gvimrc、.exrc或.vimrc，如果找到的文件不是前面列出的文件之一（也就是说这个文件尚未被当成初始化文件，也还没被执行），则当作信息来源。

警告：在 Unix 的环境中，在本地目录中包含配置文件（包括.gvimrc与.vimrc）会有安全问题，且gvim默认为设置secure选项，强制限制能从这些文件执行的事项，如果文件属于用户的话。如此有助于避免恶意代码造成破坏。如果想确认，请在你的.vimrc或.gvimrc 文件中显式设置secure选项。

使用鼠标

鼠标在gvim中于每种编辑模式中均有作用。让我们看看在标准Vim编辑模式以及gvim在两种模式中如何使用鼠标：

命令模式

于窗口底端输入冒号（:）打开命令缓冲区后即进入这个模式。如果窗口处于命令模式，可使用鼠标重置光标于命令行的任何位置。默认已启用这个模式，亦可在mouse选项中加入标志c。

插入模式

这是输入文本的模式。如果点击一个处于插入模式的缓冲区，鼠标可将光标移到选中的位置并从该处开始编辑文本。默认已启用这个模式，亦可于mouse选项中加入标志i。

鼠标在插入模式里提供简单直观、点击即可移动位置的方式。尤其是不需跳出插入模式，使用移动命令或其他方式移动后，还要再切换为插入模式。

表面看来这似乎是个好方法，但实际上只对一部分用户有益。对于熟悉Vim的用户而言，它可能弊大于利。

假设你原本在插入模式，后来暂离gvim以使用其他应用程序。当你再回头点击gvim窗口时，点击的位置就是插入文本的位置，而且大概不会是预期的位置。在单一窗口的 gvim会话中，你可能会落在与原来工作时不同的位置；在多窗口的gvim 屏幕中，你的鼠标可能点击到完全不同的窗口，文本内容可能会输入到完全不同的另一个文件中。

正常模式

包括任何不在插入模式或命令模式的时刻。在屏幕上点击鼠标，只会把光标移到选中的字符上。默认已启用这个模式，亦可于mouse选项中加入标志n。

正常模式提供直观且简单的光标放置方式，但若移动超过可视窗口的顶端或底端，则其支持相当笨重。点击并按住鼠标左键不放，然后往窗口顶端或底端拖动，gvim将相应向上或向下滚动。如果滚动停止，则把鼠标移出窗口顶端或底端，以便滚动继续（尚不清楚为何正常模式如此行动）。

正常模式中的另一个缺点，是使用户（尤其是初学者）变得依赖鼠标点击作为位置选择的方式。这点可能阻止他们积极学习Vim的移动命令，进而不能接触这种威力强大的编辑方式。最后一点，它与插入模式具有相同的点击困扰。

另外，gvim也提供可视模式（visual mode），又称选择模式（select mode）。默认已启用这个模式，亦可于mouse选项中加入标志v。可视模式是最广泛适用的模式，它让我们以拖动鼠标的方式选择文本，并高亮显示已选择的部分。使用可视模式时，能结合使用命令、插入与正常模式。

可于mouse选项中指定任意标志的组合。请参考下例所示的语法：

`:set mouse=""`
: 禁用所有鼠标行为。

`:set mouse=a`
: 启用所有鼠标行为。

`:set mouse+=v`
: 启用可视模式（v）。本例使用+=语法，把新标志加入当前的mouse设置中。

`:set mouse-=c`
: 在命令模式中禁用鼠标行为（c）。本例使用-=语法，从当前的mouse设置中删除指定标志。

初学者或许喜欢“开启”各个模式设置，但专家则可能把鼠标功能全都关闭（例如本章的作者）。

如果你使用鼠标，我们推荐通过gvim的:behave命令选择熟悉的鼠标行为，此命令接受mswin或xterm作为参数。如参数名称所示，mswin将设置为模仿Windows的行为，xterm则将设置为模仿X Window System上的一个窗口。

Vim还有几个相关的鼠标选项，包括mousefocus、mousehide、mousemodel、selectmode。可参考Vim的内置说明文档，以取得这些选项的详细信息。

如果你的鼠标带有滚轮，gvim的默认值已恰当处理它的行为，它能用于上下滚动屏幕或窗口，不管mouse选项如何设置。

有用的菜单

gvim从GUI环境带入了一项不错的功能——菜单动作，简化了一些Vim较不为人知的命令。本节特别介绍其中两种菜单。

gvim的Window菜单

gvim的*Window*（窗口）菜单包括许多最有用且最常见的Vim窗口管理命令：如分割单一GUI 窗口为多个显示区域的命令。各位有可能会觉得“撕下”（tearing off）这个菜单很方便，如图13-1所示，如此即可便利地打开窗口并切换到不同窗口。“撕下”后的效果如图13-2所示。

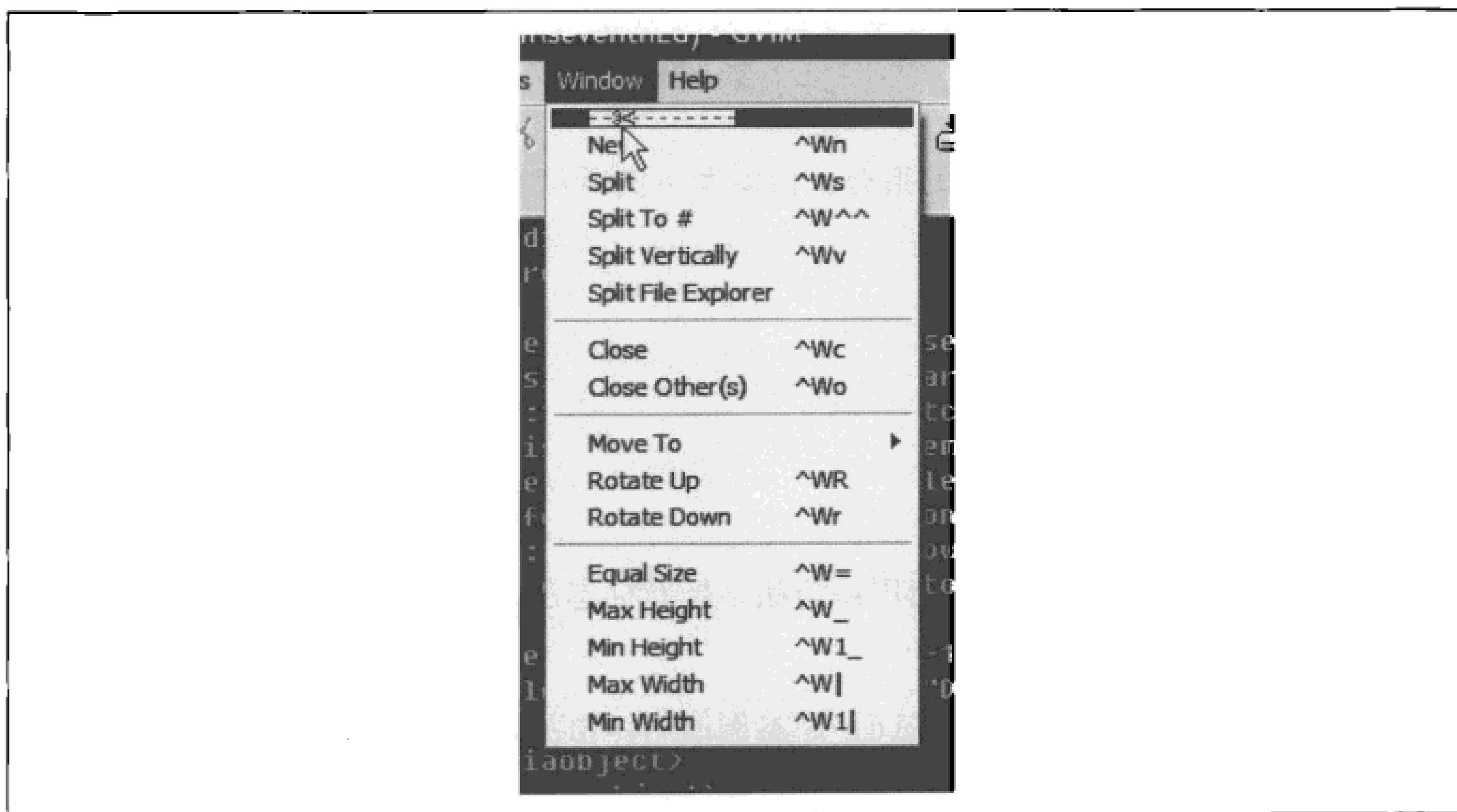


图13-1：gvim的Window菜单

gvim的右键菜单

当我们在编辑中的缓冲区里按鼠标右键时，gvim会弹出如图13-3所示的菜单。

如果选择（高亮显示）了任何文本，则会于按右键时弹出另一种菜单，如图13-4所示。

请注意，图13-3的菜单浮动时覆盖到完全无关的应用程序上。如此可让一个常用菜单随时可得，又不会阻碍编辑。对于常用的选择、剪切、复制、删除、粘贴等操作，使用这

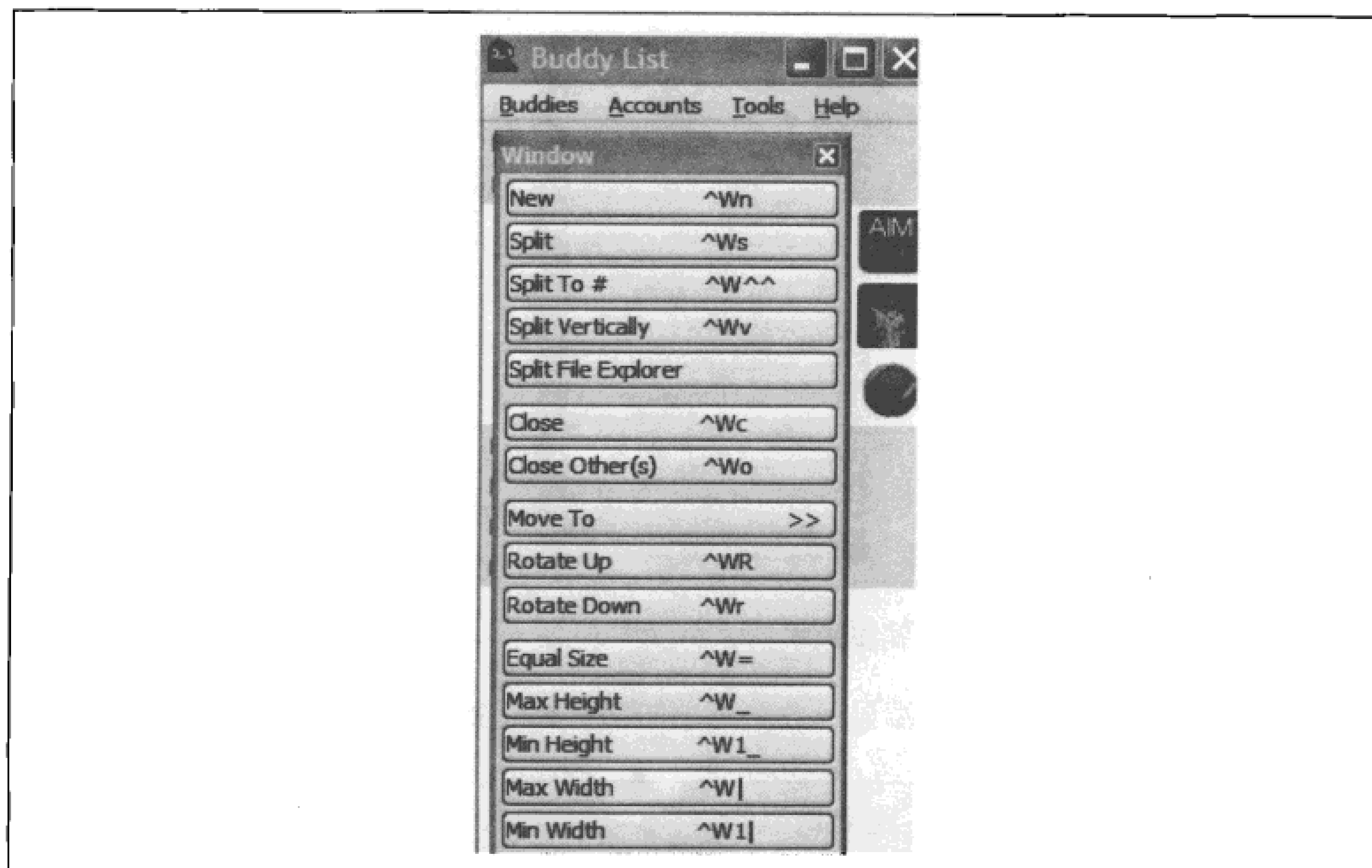


图13-2：gvim的窗口菜单，撕下并浮动的效果

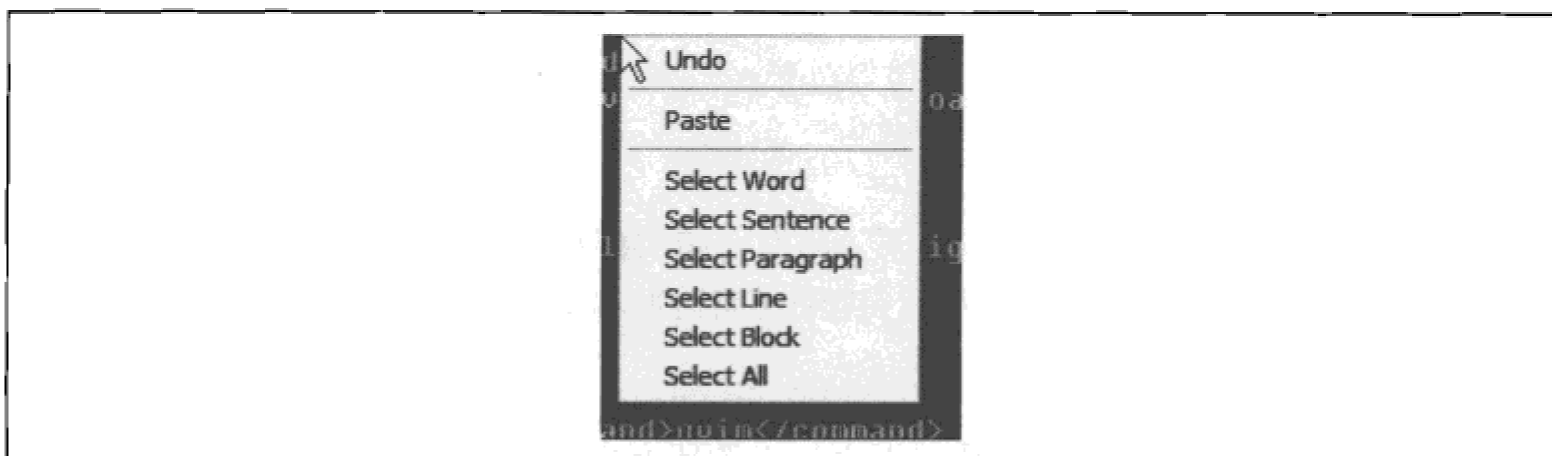


图13-3：gvim一般的编辑菜单

两种菜单都很方便。其他图形用户界面编辑器的用户已经随时都在利用这类功能，但对于长期使用Vim的用户来说，这类功能也一样好用。它在与Windows剪贴板以可预测的方式交互时尤其好用。

自定义滚动条、菜单与工具栏

gvim提供常见的GUI窗口小部件（widget），例如滚动条、菜单与工具栏。与多数时下的GUI应用程序一样，可以自定义这些窗口小部件。

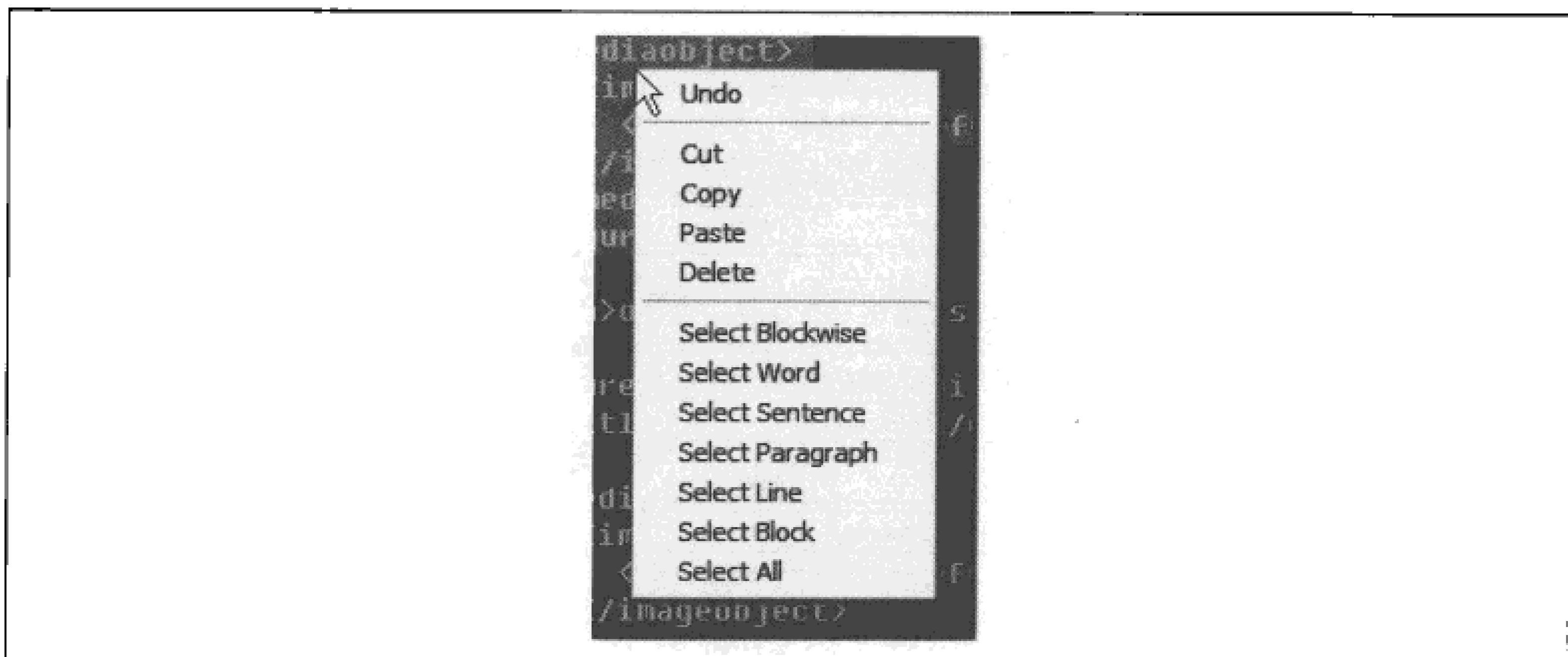


图13-4：当文本被选中时的gvim编辑菜单

默认情况下，gvim窗口于其顶端显示数个菜单与工具栏，如图13-5所示。

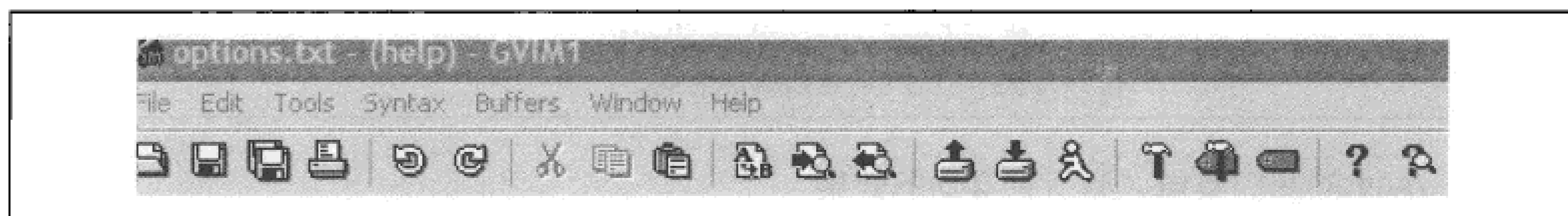


图13-5：gvim窗口的顶端

滚动条

滚动条，可在文件中快速地向上、向下、向左、向右移动，在gvim中是个可选项目。使用guioptions即可显示或隐藏滚动条，详细说明请见本章第239页的“GUI选项与命令概要”。

有趣的是，因为Vim的标准行为是列出文件中的所有文本（若有需要，则在窗口里自动让文本换行），水平滚动条在gvim会话的典型配置中没有作用。

左右滚动条的开启与关闭，则通过在guioptions选项里加入（或排除）r或l而达成。l确保屏幕上总是出现左侧滚动条，r则确保总是有右侧滚动条。大写的L或R，则要求gvim只在有垂直分割窗口时显示左右滚动条。

水平滚动条的开启与关闭则是在guioptions选项中加入或排除b来控制。

还有，你可以同时滚动左侧与右侧的滚动条！更精确地说，滚动其中一侧的滚动条，会让另一侧的滚动条跟着滚动。同时配置两侧滚动条，可以非常便利。根据你的鼠标位置，只要点击并拖动最靠近的滚动条就可以了。

注意：许多选项，包括`guioptions`在内，控制了许多行为，因此默认情况下即可容纳许多标志。未来的gvim甚至可以增加标志。因此，在`:set guioptions`命令中使用`+=`与`-=`很重要，它可避免误删需要的行为。举例来说，`:set guioptions+=l`能为gvim增加“滚动条永远在左侧”的选项，让其他guioptions选项里的内容完整无缺。

菜单

gvim有完整的自定义菜单功能。本节将讨论默认情况下的菜单（请参考前面的图13-5），并介绍如何控制菜单布局的方式。

图13-6即为使用菜单的范例，本例为Edit（编辑）菜单中的“Global Settings”（全局设置）。

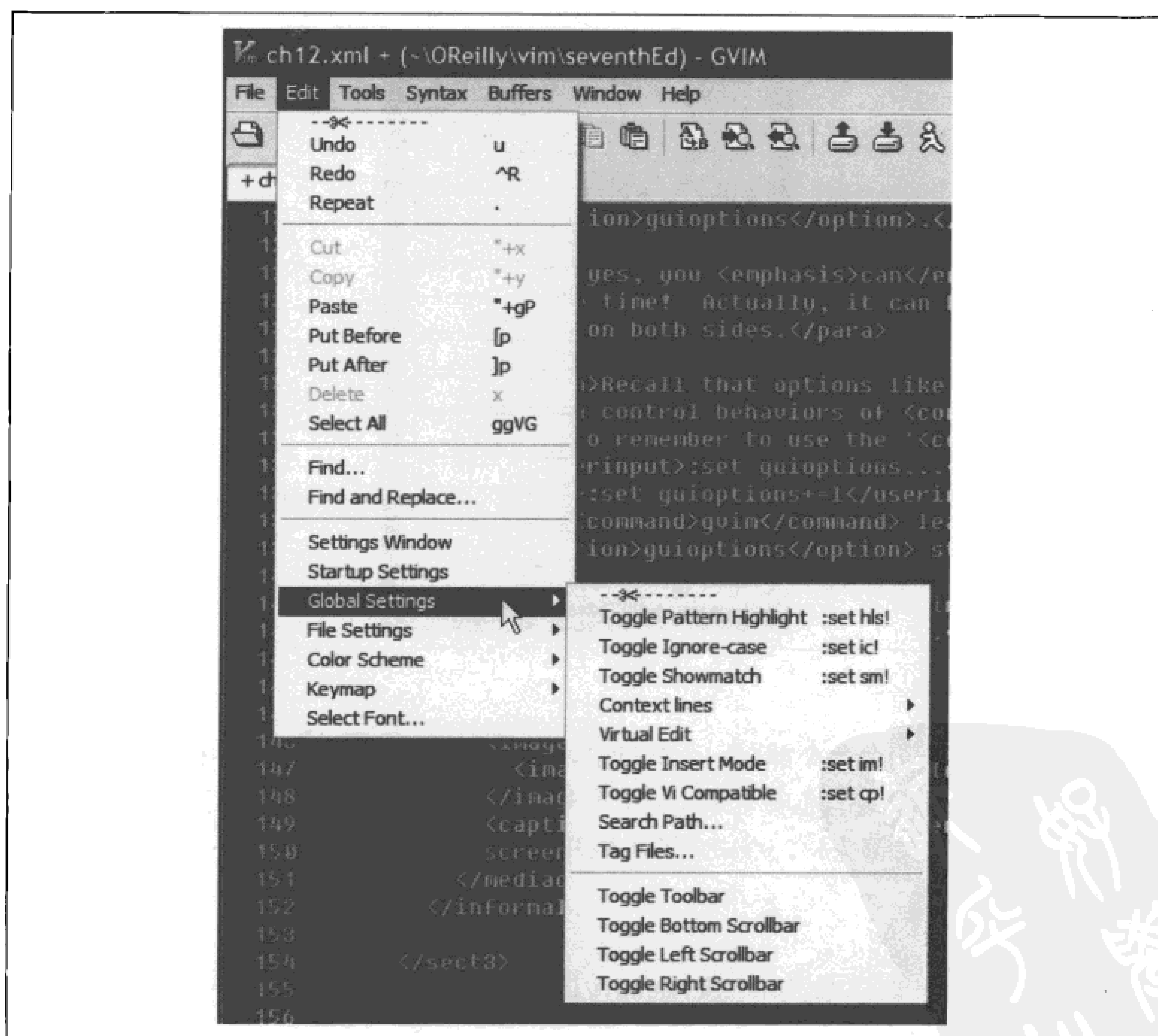


图13-6：层叠Edit菜单

注意到这些菜单选项只是Vim命令的“圆形化版”了吗？这点很有意思。事实上，我们创建与自定义菜单项的方式也是如此，稍后很快就会讨论这点。

注意：如果你注意观察菜单，包括它在右侧列出的快捷键或命令，终将慢慢学会Vim命令。以图13-6为例，虽然初学者觉得在Edit菜单中寻找熟悉的“撤销”（Undo）命令比较简单，因为它也出现在其他受欢迎的应用程序里；但在Vim里，使用快捷键u会快得多，这项信息也显示在菜单里。

如图13-6所示，每个菜单的最顶端都有一条虚线加上剪刀的图示。点击这条线即可“撕下”菜单，这样不用通过菜单工具栏，即可取用子菜单中的选项。如果你点击图13-6中“Toggle Pattern Highlight”（切换模式高亮显示）上的虚线，则将得到类似图13-7的浮动菜单。浮动菜单能放在计算机桌面上的任何地方。

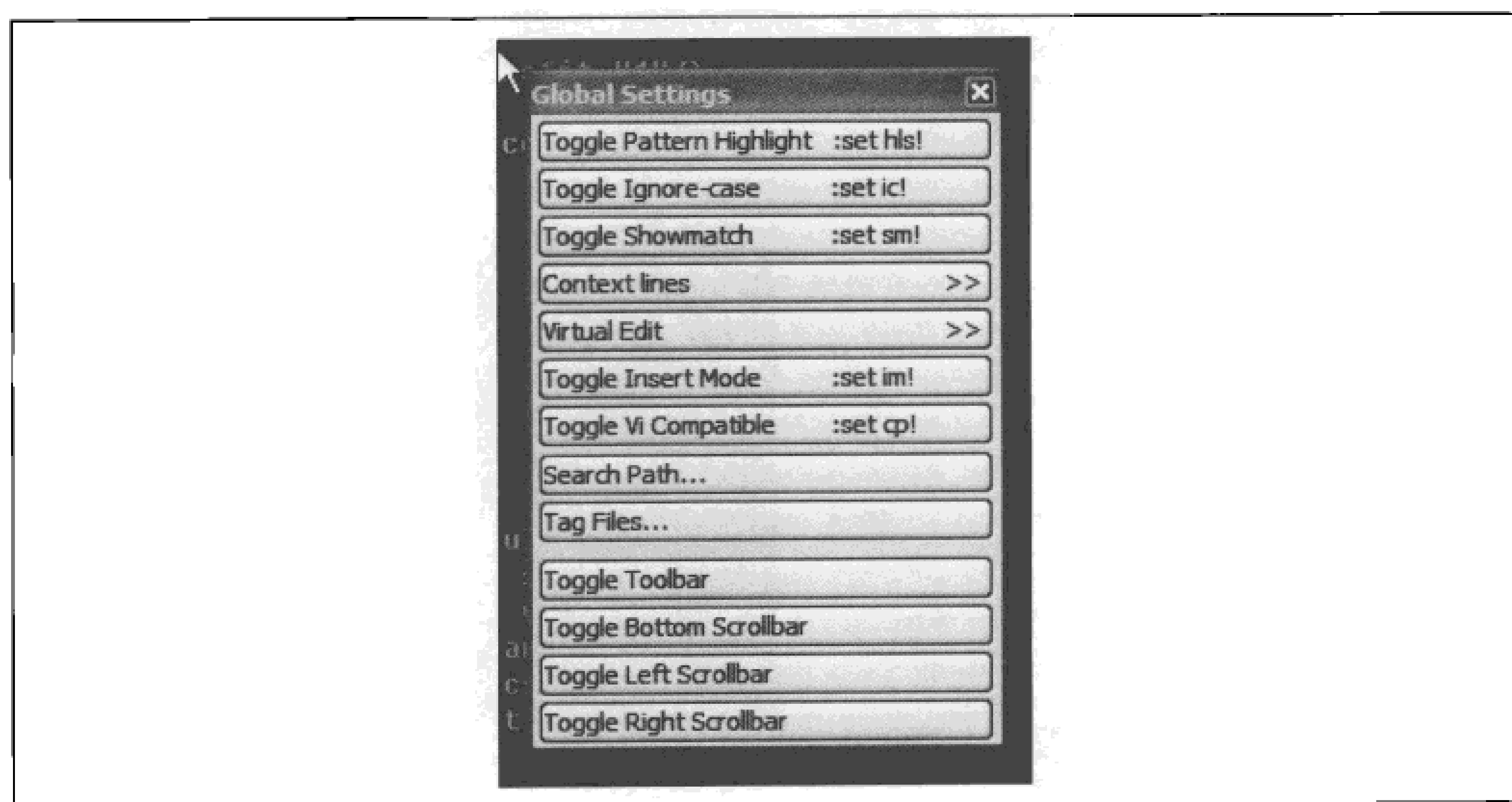


图13-7：撕下的菜单

现在，子菜单上的所有命令都在一个窗口中，只要点击即可使用。菜单中的每个选项都有自己的按钮，如果某个选项本身又是子菜单，则它的按钮右侧会有大于符号（看起来感觉像是指向右方的箭头），点击它即可展开子菜单。

基本的自定义菜单

gvim把菜单定义存储在`$VIMRUNTIME/menu.vim`文件里。

定义菜单项的方式与“映射”（mapping）类似。如第108页的“使用map命令”一节所述，以下即为映射某个键的方式：

```
:map <F12> :set syntax=html<CR>
```

菜单也采用很相似的方式。

假设这次不是要把F12映射到“设置语法为`html`”的行为，而是想在File菜单里增加专门的“HTML”项进行此项任务。请使用`:amenu`：

```
:amenu File.HTML :set syntax=html<CR>
```

`<CR>`请如上例所示一样逐字输入，以作为命令的一部分。

现在检查你的File菜单。应该出现了新的HTML项，如图13-8所示。通过使用`:amenu`（而非`menu`），可确保该项能在所有模式（命令、插入、正常模式）中取用。

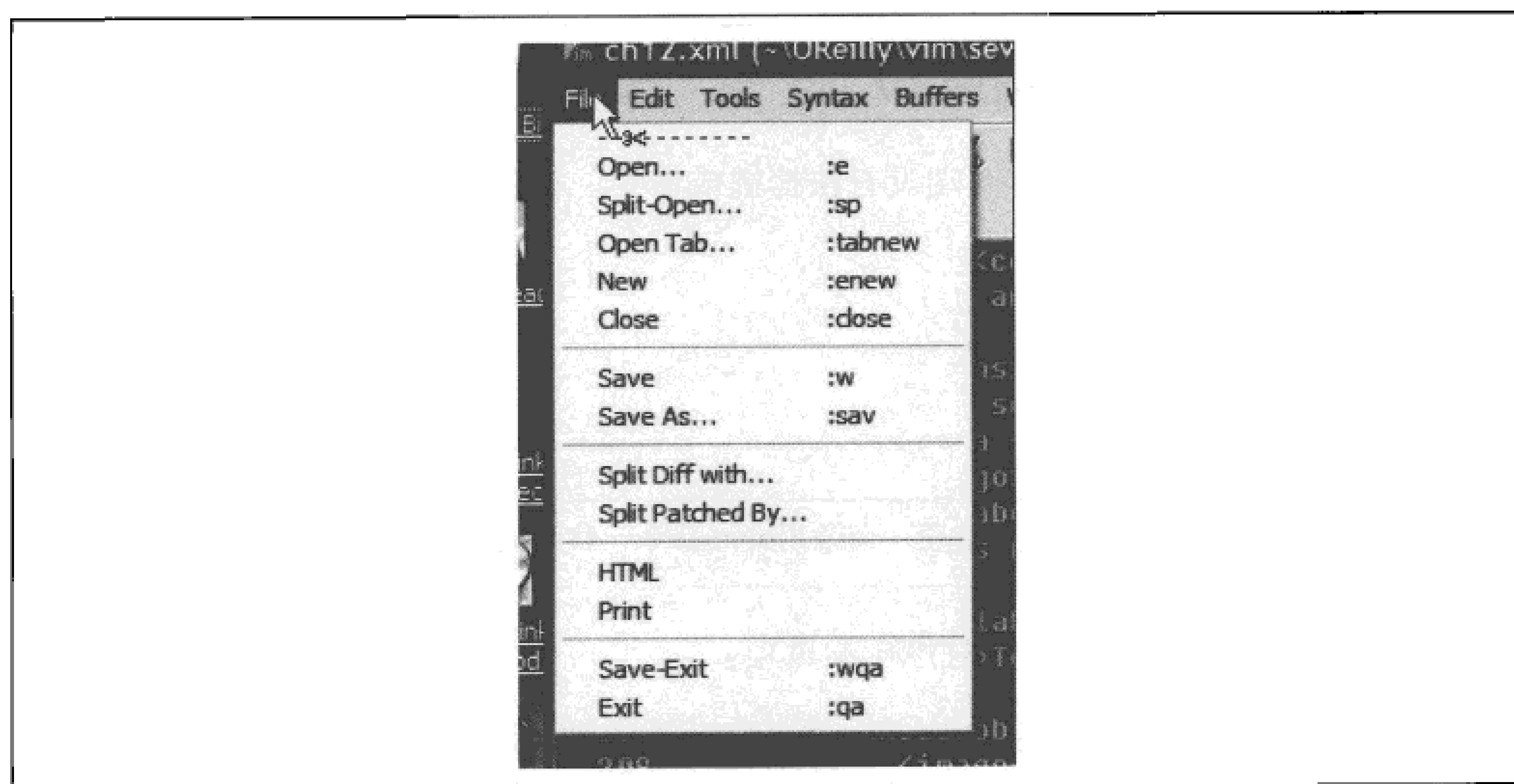


图13-8：File菜单下的“HTML”菜单项

注意：`menu`命令加入的菜单项只有在命令模式中能用，即该项不会出现在插入与正常模式中。

菜单项的位置是由一系列以点号（`.`）分隔的层叠（`cascading`）菜单项指定的。本例中，`File.HTML`把菜单项“HTML”放在File菜单。这系列中的最后一项即为欲加入的新选项。当前只是在现有菜单中新增选项，但很快就会看到创建一个全新的层叠菜单也很简单。

请务必测试新菜单项。例如编辑一个Vim检测为XML的文件，可于类似图13-9的状态行中看到识别结果。我们已经把Vim或`gvim`的状态行设置为显示当前启用的语法（于状态行最右侧）。（请参考第205页的“不错的Vim取巧诀窍”一节。）

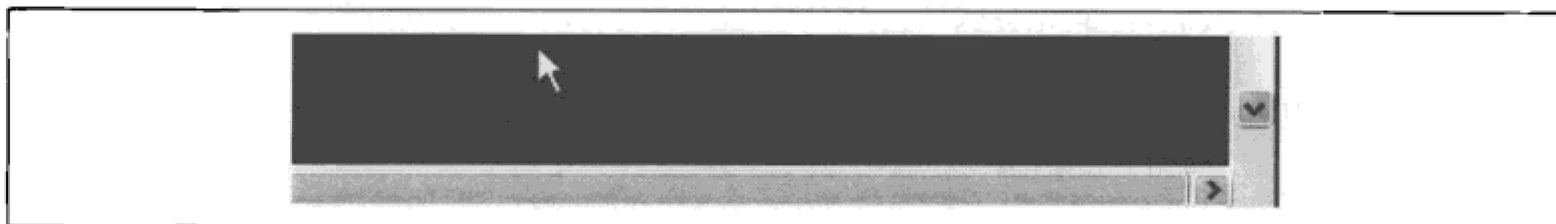


图13-9：新菜单动作前状态行显示XML语法

调用新的HTML菜单项后，Vim状态行会检查菜单项能否运作，语法也改为HTML，如图13-10所示。



图13-10：新菜单动作后状态行显示HTML语法

我们加入的HTML菜单项，其右端没有快捷键或命令。让我们重做一次新增菜单项，加入这项强化功能。

首先，删除现有的HTML菜单项：

```
:aunmenu File.HTML
```

注意： 如果只在命令模式中新增菜单项，请改用menu命令，移除菜单项则使用unmenu。

下一步，新增显示相对应命令的HTML菜单项：

```
:amenu File.HTML<TAB>syntax=html<CR> :set syntax=html<CR>
```

菜单项规范加入了<TAB>与syntax=html<CR>。通常，要在菜单的右端显示文本，请把syntax=html放在符串<TAB>后并于结尾处加上CR。图13-11显示了修改后的 File 菜单。

注意： 如果想在菜单项的描述文字（或菜单名本身）中使用空格，请在空格前加上反斜线（\）。如果不用反斜线，Vim将使用第一个空格字符后的一切内容作为菜单动作的定义。就前例而言，如果想采用:set syntax=html而非syntax=html，则:amenu命令应修改为：

```
:amenu File.HTML<TAB>set\ syntax=html<CR> :set syntax=html<CR>
```

大多数情况中，最好不要修改默认的菜单定义，而是改为创建独立的菜单项。虽然需在根层次定义新菜单，但这与新增菜单项到现有菜单中一样简单。

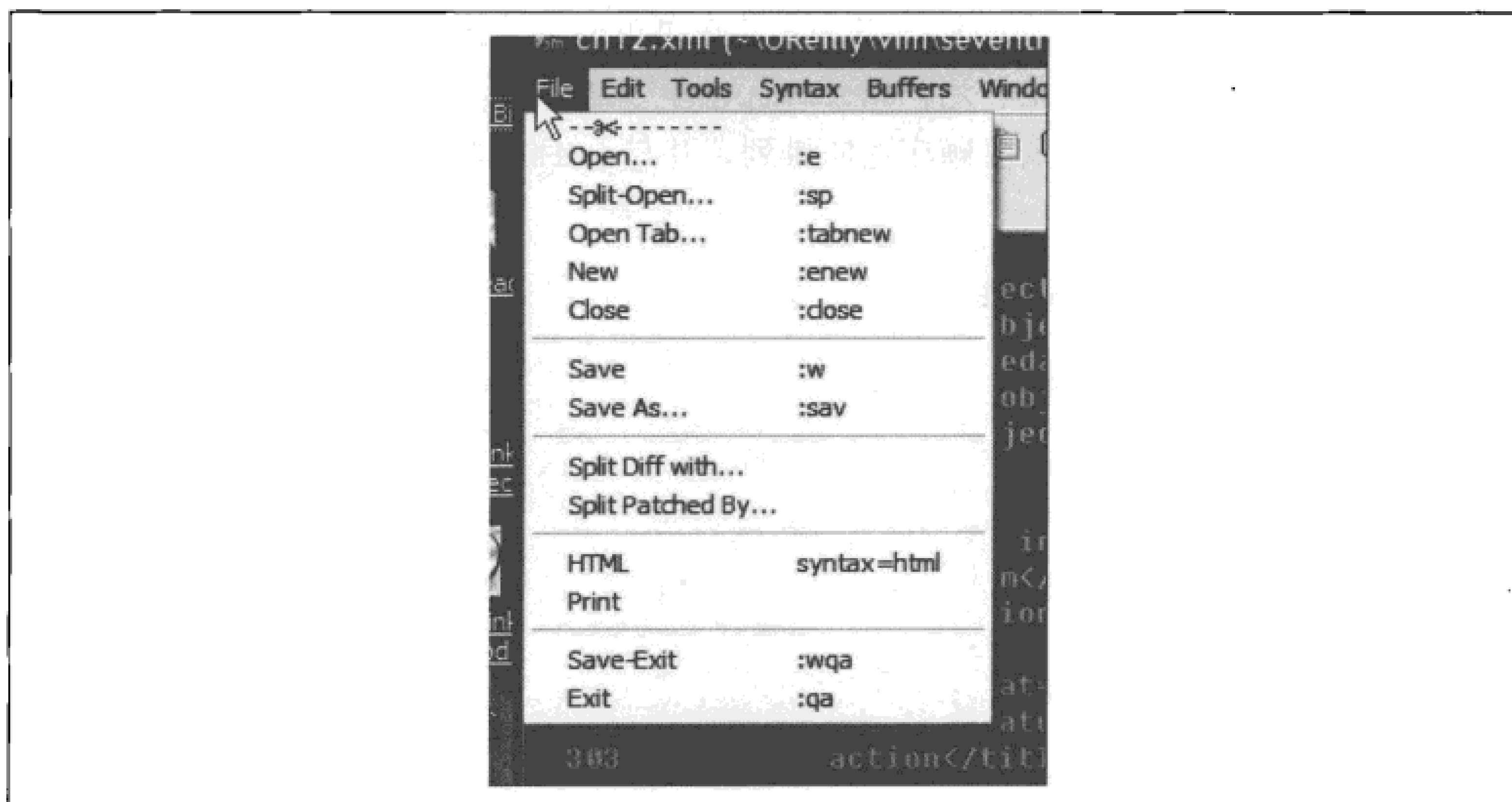


图13-11：HTML菜单项，显示相关联的命令

继续使用前面的范例，让我们在菜单栏上创建新的菜单MyMenu，而后在新菜单中增加HTML菜单项。首先，从File菜单中移除HTML项：

```
:aunmenu File.HTML
```

接着输入以下命令：

```
:amenu MyMenu.HTML<TAB>syntax=html :set syntax=html<CR>
```

图13-12显示了菜单栏的改变。

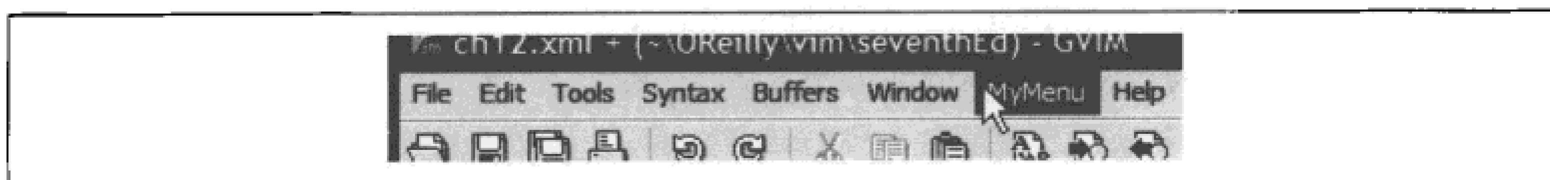


图13-12：添加了“MyMenu”菜单后的菜单栏

与菜单相关的命令对于菜单出现的位置及其行为提供更多细部控制，例如命令是否指示任何活动，甚至包括菜单项是否显示。我们将于后续章节中进一步讨论这些可能性。

再谈自定义菜单

在见识过修改与扩展gvim的菜单有多容易后，让我们一起看看更多自定义与控制菜单的范例。

前面的范例并未指定放置新增的“MyMenu”菜单的位置，gvim默认把它放在Window（窗口）与Help（帮助）中间。gvim能根据菜单所注明的优先级控制其放置位置。优先

级只是指派给每个菜单的一个数值，以决定其在菜单栏上的排列位置。数值越大，菜单的位置就越靠右端。不幸的是，用户猜想的优先顺序刚好与gvim的定义相反。想直接看出优先顺序，请把图13-5的菜单顺序逆向排列，并与gvim的默认菜单顺序比较，请见表13-1所示。

表13-1：gvim 的默认菜单优先级

菜单名称	优先级
File（文件）	10
Edit（编辑）	20
Tools（工具）	40
Syntax（语法）	50
Buffers（缓冲区）	60
Window（窗口）	70
Help（帮助）	9999

大多数用户都以为File的优先级高于Help（所以File才位于最左端，而 Help 位于最右端），但其实Help的优先级比较高。所以，请把优先级值理解成菜单靠近右边界的顺序。

在menu命令前加上数值，即可定义菜单的的优先级。如果没有指定任何值，默认值为500，因此前例中的MyMenu才会出现在那个位置：位于Window（70）与Help（9999）间。

假设我们的新菜单要在File与Edit间，则需指定一个大于10、小于20的值给 MyMenu。下列命令指派优先级值为15，这可达成我们想要的效果：

```
:15amenu MyMenu.HTML<TAB>syntax=html :set syntax=html<CR>
```

注意： 菜单出现后，它在整个编辑会话中的位置就已固定，而且不会为了试图影响菜单优先级的命令而改变位置。例如，下达新增菜单项的命令，但在命令前加上不同于菜单优先级的值，菜单的位置不会因此而改变。

若想把菜单位置与优先级弄得更混乱，我们还可以通过指定优先级来控制菜单项在菜单中的位置。优先级高的菜单项比优先级低的菜单项处于菜单的更底端，但其语法与定义菜单优先级的语法不同。

我们扩展前面的一个菜单范例，为其中的HTML菜单项指定非常高的优先级值（9999），让它显示在File菜单的底端：

```
:amenu File.HTML .9999 <TAB>syntax=html<CR> :set syntax=html<CR>
```

为什么要在9999前有点号？因为此处需要指定两种优先级，故以点号分隔：一个优先级赋予File，另一个赋予HTML。我们空下File的优先级不写，因为它是个已存在的菜单，不能改变。

一般而言，菜单项的优先级出现在该项所在的菜单与菜单项的定义间。对菜单层次结构中的每一层都必须指定优先级，或以点号表示空下该层次的优先级。因此，增加一个层次结构中很深的菜单项时，例如"Edit →Global Settings→ Context lines→Display"，且想把最后一个菜单项（Display）的优先级设为 30时，则指定方式需为`...30`，菜单项位置加上优先级的表示方式如下所示：

```
Edit.Global\ Settings.Context\ lines.Display ...30
```

与菜单优先级一样，指派菜单项的优先级后，其即固定。

最后，还可用`gvim`的菜单分隔命令控制菜单的"留白"。使用与新增菜单项相同的定义，只是把命令名称中的"..."改为前后加上连字符（-）。

综合运用

现在我们已经知道了如何创建、放置与自定义菜单了。让我们把前面讨论过的命令加入.gvimrc文件中，变成gvim环境的永久成员。命令的顺序应该大致与下例相同：

```
" add HTML menu between File and Edit menus
   ❶15amenu MyMenu.XML<TAB>syntax=xml :set syntax=xml<CR>
   ❷amenu  ❸.600 MyMenu.-Sep- :
   ❹amenu  ❺.650 MyMenu.HTML<TAB>syntax=html :set syntax=html<CR>
   ❻amenu  ❼.700 MyMenu.XHTML<TAB>syntax=xhtml :set syntax=xhtml<CR>
```

现在，我们有了位于顶层的个性化菜单，可快速取用上述三个最爱用的命令。关于上例，有些重点需要注意：

- 第一个命令（❶）使用 `15` 为前缀，告诉 `gvim`所使用的优先级为 15。对于未自定义的环境，这个值将把新菜单放在 File 与 Edit 菜单中间。
- 接下来的命令（❷、❹、❻）并未指定优先级，因为一旦决定了一个优先级，就不会使用其他值了。
- 我们在第一个命令后，使用子选单优先级语法（❸、❺、❼）以确保新菜单项的顺序正确。请注意，我们的第一个命令定义为`.600`。这用于确保子菜单项均列在第一定义的菜单项下，由于我们并未指定它的优先级，因此它采用默认值500。

为了访问方便，请单击"剪刀"图标以撕你的个性化浮动菜单，如图13-13所示。

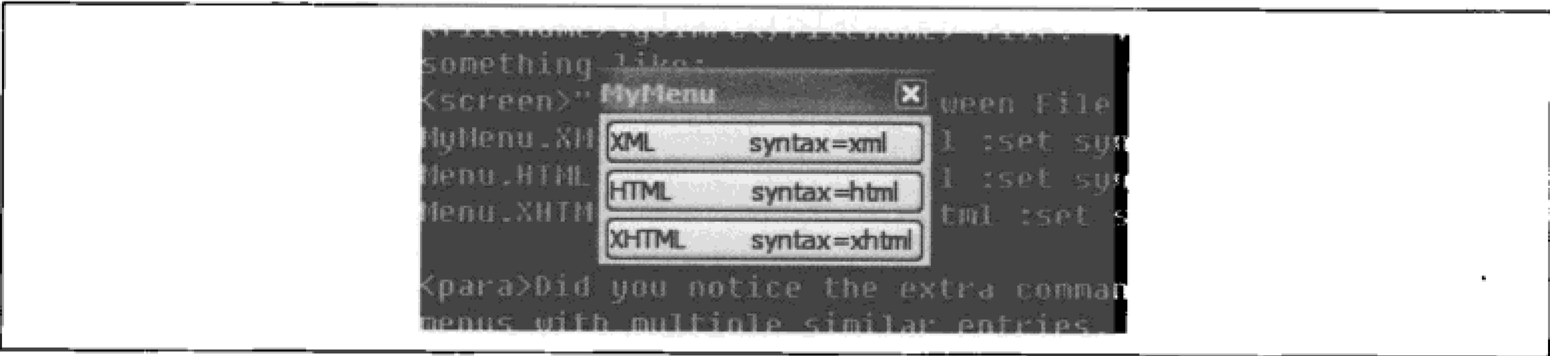


图13-13：个性化浮动的撕下菜单

工具栏

工具栏是列了许多图标的长条形面板，使用户能快速使用程序功能。例如在 Windows 上，gvim即于窗口顶端显示了图13-14所示的工具栏。

表13-2列出了工具栏图标及其意义。

表13-2：gvim 工具栏图标及其意义

图示	说明	图示	说明
	打开文件的对话框		查找下一个搜索模式
	保存当前编辑的文件		查找前一个搜索模式
	保存所有文件		选择欲载入已保存的编辑会话
	打印缓冲区		保存当前的编辑会话
	撤销前次更改		选择欲运行的Vim脚本
	重做前次动作		用make命令编译当前项目
	剪切选择的区域至剪贴板		为当前的目录树建立标签
	复制选择的区域至剪贴板		跳至光标处的标签
	粘贴剪贴板中的内容至缓冲区	?	打开帮助
	查找并替换		搜索帮助

如果这些图标看起来不熟悉或不够直观，你可以让工具栏同时显示文字与图标。请执行如下命令：

```
:set toolbar="text,icons"
```

注意：与许多Vim的高级功能一样，工具栏功能需于编译时选择打开。如果用户不需要工具栏，则可排除这个功能以节省内存。只有+GUI_GTK、+GUI_Athena、+GUI_Motif+GUI_Photon功能被编译进你使用的gvim版本里，才会出现工具栏。第九章介绍了如何重新编译Vim，可创建对gvim可执行文件的链接。

调整工具栏的方式与调整菜单的非常相似。事实上，我们使用相同的:menu命令，只是加上图片专用的语法。虽然有个帮助gvim寻找各命令相应图标的算法，我们还是明确地指向欲采用的图标图片。

gvim把工具栏当成一维数组。而且，与我们控制新菜单由右向左排列的方式一样，也可以控制新工具栏项的位置，其方法是在menu命令前加上指定位置优先顺序的数值。但是，没有创建新工具栏的表示方式。所有新工具栏的定义出现在单一工具栏上。新增工具栏的语法片段为：

```
:amenu icon=/some/icon/image.bmp ToolBar.NewToolBarSelection   Action
```

其中，*/some/icon/image.bmp*表示包含工具栏按钮或图像（通常是图标）的文件路径，这样图标才能显示在工具栏上；*NewToolBarSelection*是新的工具栏按钮；*Action*定义按钮的动作。

举例来说，我们定义一个新的工具栏选项，该选项被单击或选择时，将在Windows中打开DOS窗口。假设Windows路径设置正确（应该正确），我们将定义一个通过执行如下命令（即其Action）在gvim中打开DOS窗口的工具栏选项：

```
:!cmd
```

对于新选项的工具栏按钮（或图标），我们使用表示DOS命令提示符的图标，如图13-15所示，系统把它存储在$HOME/dos.bmp中。

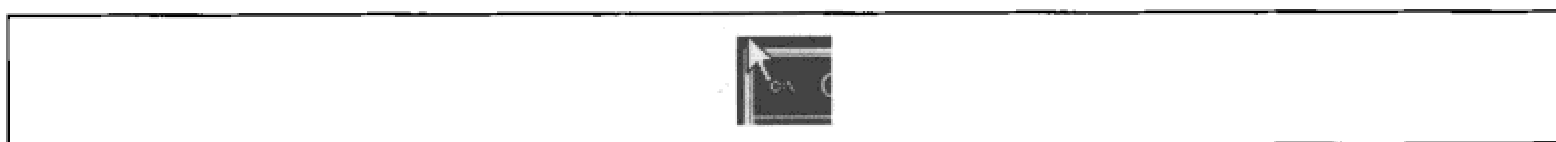


图13-15：DOS图标

执行命令：

```
:amenu icon="c:$HOME/dos.bmp" ToolBar.DOSWindow :!cmd<CR>
```

即可创建工具栏项并把图标加入工具栏末端。现在工具栏应如图13-16所示。新图标出现在工具栏的最末端。

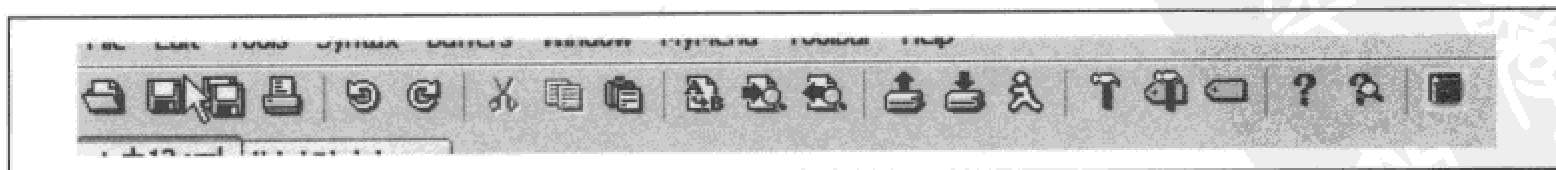


图13-16：增加了DOS命令的工具栏

工具提示

gvim可定义菜单栏项与工具栏图标的工具提示（tooltips）。菜单的工具提示是在鼠标移过该选项时显示于gvim命令行区域。工具栏的工具提示是在鼠标移过工具栏图标时跳出说明框。如图13-17所示，当我们把鼠标移到工具栏的“Find Previous(寻找前一个）”按钮时，跳出了工具提示。

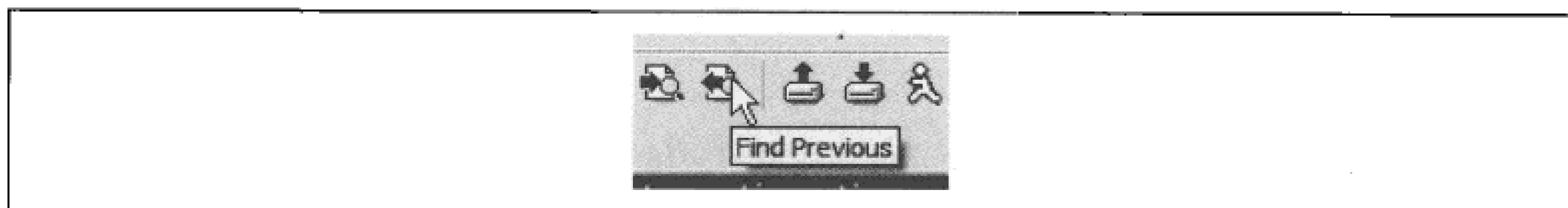


图13-17：Find Previous图标的工具提示

:tmenu命令负责为菜单与工具栏选项定义工具提示。语法为：

```
:tmenu TopMenu.NextLevelMenu.MenuItem tool tip text
```

*TopMenu.NextLevellMenu.MenuItem*用于定义我们加入工具提示的选项的层次（从最顶端开始计算直到选项所在位置）。例如为File菜单下的Open菜单项定义工具提示，将用如下命令：

```
:tmenu File.Open Open a file
```

如果要定义的工具提示的对象为工具栏项，则使用ToolBar作为最顶层“菜单”（工具栏没有真正的最顶层菜单）。

让我们为工具栏的DOS工具栏图标（于前一节创建的范例）定义弹出式的工具提示。请输入如下命令：

```
:tmenu ToolBar.DOSWindow Open up a DOS window
```

现在鼠标经过新增加的工具栏图标时，即可看到工具提示，如图13-18所示。

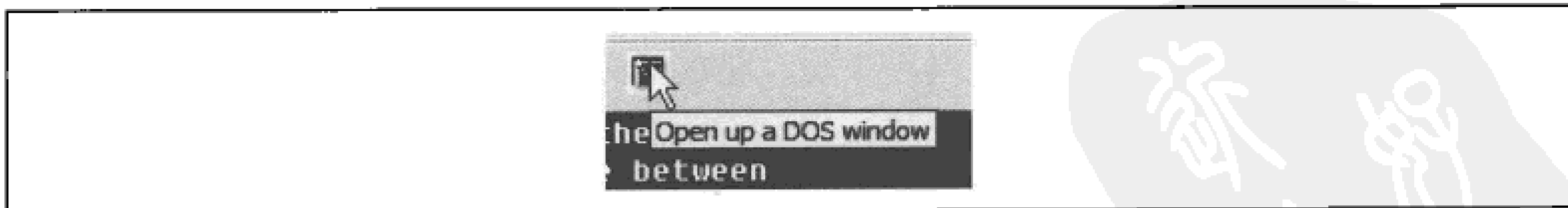


图13-18：增加了DOS命令及其新的工具提示的工具栏

Microsoft Windows中的gvim

gvim逐渐受到Windows用户的欢迎。熟练的vi与Vim用户也将发现 Windows 的gvim版本非常好用，并且大概是众多平台上可采用的最新版本。

注意： 可自安装的可执行文件应能自动且天衣无缝地集成Vim到Windows环境中。如果没有，请查阅Vim运行时目录下的gui-w32.tex帮助文件以了解regedit提示。因为修正工作需编辑Windows Registry（注册器），只要你对这个程序有一丝一毫地没有把握，请别尝试修改它。你或许能找到更擅长注册器的人帮助。这是个常见但很重要的练习尝试。

长期使用Windows的用户都很熟悉剪贴板——一个保存文本与其他信息的地方，帮助实现复制、剪切、粘贴等操作。Vim支持与Windows剪贴板的交互。只要在可视模式中高亮显示文本，并点击菜单项Copy或Cut，即可存储Vim中的文本至Windows剪贴板。然后可将文本粘贴到其他Windows应用程序中。

X Windows System中的gvim

习惯使用X环境的用户可以定义并使用许多可调整的X功能。例如，利用通常定义在.Xdefaults文件中的标准类定义可定义许多资源。

警告： 请注意这些标准X资源（X resource）只适用于Motif或Athena版的GUI。显然，Windows 版的不会了解X资源，甚至KDE或GNOME也未支持 X 资源。

关于X以及如何配置并自定义它的完整讨论已在别处有详尽的说明，而且这也超出本书的讨论范围。可参考 X 说明手册中（不那么简短）的概述。

GUI选项与命令概要

表13-3总结了专门与gvim相关的命令与选项。当Vim编译时加入对GUI的支持，这些命令或选项即会被加入Vim，并在调用gvim或vim -g时发挥效用。

表13-3：针对gvim的选项

命令或选项	类型	说明
guicursor	选项	设置光标的形状与闪烁
guifont	选项	使用的单字节字体的名称
guifontset	选项	使用的多字节字体的名称
guifontwide	选项	双字节字符采用的字体名称
guiheadroom	选项	留给窗口装饰用的像素数量
guioptions	选项	使用的组件与选项
guipty	选项	为“:!”命令使用pseudo-tty

表13-3：针对gvim的选项（续）

命令或选项	类型	说明
guitablabel	选项	为分页自定义标签
guitabtooltip	选项	为分页自定义工具提示
toolbar	选项	显示在工具栏中的条目
-g	选项	打开GUI（也可允许其他选项）
-U gvimrc	选项	打开GUI时使用gvim启动文件，命名为grimrc或类似名称
:gui	命令	打开GUI（仅限类似UNIX的系统）
:gui *filename*...	命令	打开GUI并编辑指定文件
:menu	命令	列出所有菜单
:menu *menupath*	命令	列出所有以*menupath*开始的菜单
:menu *menupath action*	命令	新增菜单（路径为*menupath*）以执行*action*
:menu *n menupath action*	命令	新增菜单（路径为*menupath*）并附有位置优先级值*n*
:menu ToolBar.*toolbarname action*	命令	新增工具栏项*toolbarname*，以执行*action*
:tmenu *menupath text*	命令	为菜单项*menupath*创建工具提示，*text*为提示内容
:unmenu *menupath*	命令	删除*menupath*指定的菜单

第十四章

程序员专用的Vim强化功能

文本编辑只是Vim的强大套件之一。优秀的程序员需要功能强大的工具以确保工作的效率与娴熟。好的编辑器只是个起点，仅这样还不够。许多时下的编程环境均试图提供综合方案，但其实这只需要一个强大的编辑器再加上一些智慧。

编程工具提供额外功能，范围从编辑时加上语法着色（syntax coloring）、自动缩排与格式化、关键字补全……到全面成熟的集成开发环境（Integrated Development Environment，IDE），它们具有构建完整开发系统所需的精细集成。这类IDE可能非常昂贵（例如 Visual Studio），也可能完全免费（Eclipse），但它们都需要大量磁盘空间与内存，学习曲线也很陡峭，还需要无限的资源。

程序员的任务有很多种，各人的技术需求也是一样。小规模的开发任务能以简单的编辑器（提供比文本编辑稍多一点的功能）轻松完成。大型、多组件、多平台、多人员的工作则几乎能抵消IDE为我们承担的工作量。但根据传闻，许多老练的程序员觉得，IDE提供太多复杂的功能，不过对于成功的可能性却没有什么提高。

Vim在简易编辑器与庞大的IDE中间给出了一个不错的折衷方案。它具有最近才能在昂贵的IDE上使用的功能；又让我们能进行快速而简单的编程任务，而不需忍受IDE的学习曲线与消耗的成本。

特别适合程序员的选项、功能、命令与函数，包括把许多行代码折叠成一行、语法着色、自动格式化……Vim为程序员提供了许多工具，只有实际使用，才知道这些工具的好处。在高端部分，则提供了一个缩小版IDE，称为Quickfix，它对许多编程任务也是很方便的功能。本章将讨论下列主题：

- 折叠
- 自动智慧缩排
- 关键字与字典词汇补全

- 标签与扩展标签
- 语法高亮显示
- 语法高亮显示编写（自行）
- Quickfix，Vim的缩小版IDE

折叠与大纲（大纲模式）

折叠功能让我们可自定义欲看到的部分文件。例如在代码块内，你可以隐藏任何在花括号（{}）里的内容，或隐藏所有命令。折叠共有两个步骤。首先，使用任意一种折叠方式（稍后另有讨论），在构成块的文本中定义需要折叠的内容。下一步，使用折叠命令时，Vim即隐藏指定的文本并在它的位置留下一行占位符。图14-1显示了Vim的折叠效果。我们可用折叠占位符管理隐藏起来的文本行。

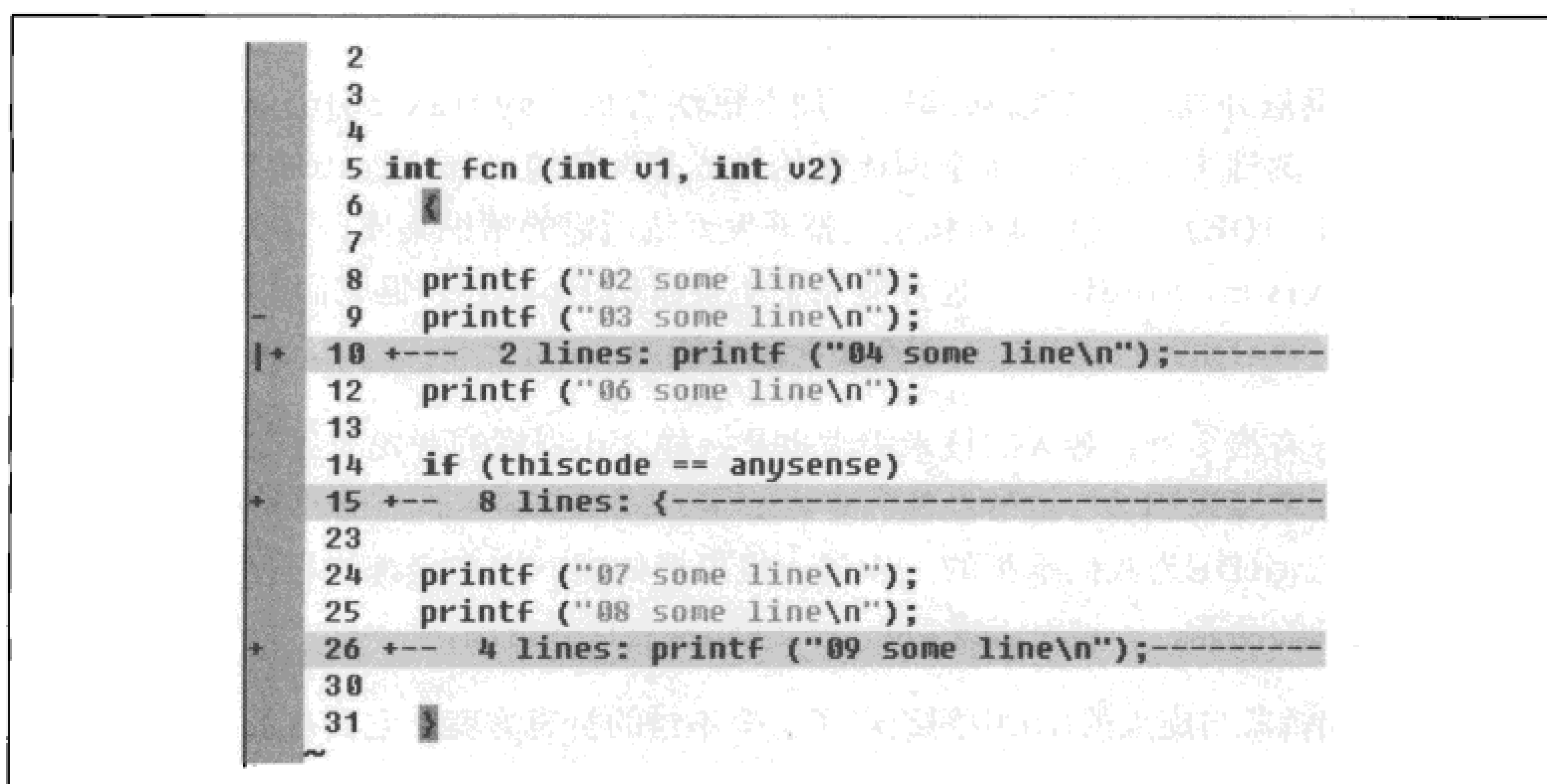


图14-1：Vim折叠范例

在范例中，第11行因为一个2行的折叠（开始于第10行）而被隐藏。第15行开始则有一个8行的折叠，隐藏起第16行到第22行。从第26行开始，则有一个4行的折叠，隐藏了第27行到29行。

对于可创建的折叠行数，实际上并无限制，甚至可创建嵌套折叠（折叠中的折叠）。

有些选项可控制Vim创建与显示折叠的方式。还有，如果你曾要花时间创建许多折叠，Vim为此提供了便利命令：`:mkview`与`:loadview`，可在会话间保留折叠，不需再次创建。

学习折叠需要些努力，但熟练后你便多了一种强大工具，可控制显示的内容与时机。别低估了这项功能带来的威力。正确且可维护的程序在许多层面都需要可靠的设计，所以良好的编程者通常需要看到整个森林而非一棵树——换句话说，为了看清文件的整体结构，往往需要忽略实现的一些细节。

对于熟练的用户，Vim提供了6种定义、创建、操纵折叠的方式。这种灵活性让我们能在不同的上下文中创建并管理折叠。最后，一旦创建了，折叠的打开、关闭与行为将与整套折叠命令相近。

创建摺叠的6种方式：

`manual`
: 以标准Vim结构定义折叠跨越的范围，类似移动命令。

`indent`
: 折叠与折叠的层次，对应于文本的缩排与`shiftwidth`选项值。

`expr`
: 以正则表达式定义折叠。

`syntax`
: 折叠对应于文件所用的程序语言语义（例如C程序的函数块即可折叠）。

`diff`
: 以两个文件间的差异定义折叠。

`marker`
: 以文件中的预定义（亦可由用户定义）标记指定折叠边界。

所有方式对折叠的控制（打开与关闭、删除等等）都一样。我们将介绍手动（manual）折叠，并详细讨论Vim的折叠命令。也会提到其他方式的一些细节，但这些细节很复杂，并有专门的使用场合，已超出本书概述的范围。我们希望这里讨论到的内容能刺激大家探讨其他方式的丰富用途。

好了，让我们简单看一下重要的折叠命令，并进入折叠的简短范例。

折叠命令

折叠命令均以z开头。这同样是为了方便大家记忆，请想想一张纸折叠后（折法正确的话），它的侧面不是有点像字母“z”吗？

大约有20个与折叠相关的命令。使用这些命令，可以创建或删除折叠、打开或关闭折叠

（显示或隐藏属于折叠中的文本）以及切换折叠的显示或隐藏状态。以下列出命令及简短的说明：

`zA`
: 递归切换折叠状态。

`zC`
: 递归关闭折叠。

`zD`
: 递归删除折叠。

`zE`
: 去除所有折叠。

`zf`
: 创建折叠，范围从当前的行开始到光标移动后到达的位置结束（借由移动命令改变光标位置）。

count`zF`
: 创建涵盖*count*行的折叠，从当前的行开始。

`zM`
: 设置 `foldlevel` 选项为 0。

`zN, zn`
: 设置（`zN`）或复位（`zn`）`foldenable`选项。

`zo`
: 递归打开折叠。

`za`
: 切换一个折叠的状态。

`zc`
: 关闭一个折叠。

`zd`
: 删除一个折叠。

`zi`
: 切换`foldenable`选项的值。

`zj, zk`
: 移动光标至下一个折叠开始的地方（`zj`），或移动至前一个折叠的结尾处（`zk`）

（请注意易记的j（“jump”）与k移动命令以及它们在折叠上下文中如何类比移动的方式）。

zm, zr

递减（zm）或递增（zr）foldlevel选项的值。

zo

打开一个折叠。

警告： 不要把删除折叠与一般删除命令弄混。使用删除折叠的命令以移除折叠或去除折叠的定义。折叠被删除，对于包含在该折叠中的文本没有影响。

---

zA、zC、zD、zO被形容为递归（recursive）的原因，在于用这些命令对某个折叠进行操作后，其内部嵌套的所有折叠也成为操作对象。

手动折叠

如果你知道Vim的移动命令，就已经精通手动折叠所需的一半知识。

以隐藏三行于折叠中为例，下列两个命令都能达成：

```
3zF
2zfj
```

3zF对三行内容执行折叠命令zF，从当前的行开始。2zfj则是从当前的行开始执行zf命令，范围到j移动光标所至的地方（本例为向下两行）。

让我们尝试更复杂的命令，以使用在C程序上为例。想折叠一块C的代码时，请把光标放在代码块的开始或结尾括号（{或}）上并输入zf%（%移动光标至对应的括号上）。

输入zfgg可创建从光标位置直达文件开始处的折叠（gg移动光标到文件的开始处）。

通过范例，可以比较容易了解折叠。我们将用一个简单文件创建并操纵折叠，同时观察折叠的行为。我们也将看到一些Vim提供的可视折叠。

首先看看图14-2的范例文件，它包含一些（无意义的）C代码。刚开始，没有折叠。

图中有些需要注意的地方。首先，Vim于屏幕左端显示了行编号。我们推荐大家一直打开显示行编号的选项（使用number选项），以增加辨别文件位置的可视信息，在本例的

```
 4
 5 int fcn (int v1, int v2)
 6   {
 7
 8   printf ("02 some line\n");
 9   printf ("03 some line\n");
10   printf ("04 some line\n");
11   printf ("05 some line\n");
12   printf ("06 some line\n");
13
14   if (thiscode == anysense)
15     {
16
17     printf ("07 some other line\n");
18     printf ("08 some other line\n");
19     printf ("09 some other line\n");
20     printf ("10 some other line\n");
21
22     }
23
24   printf ("07 some line\n");
25   printf ("08 some line\n");
26   printf ("09 some line\n");
27   printf ("10 some line\n");
28   printf ("11 some line\n");
29   printf ("12 some line\n");
30
31   }
```

图14-2：没有折叠的范例文件

上下文中，行编号的作用在部分内容被折叠后更为可贵。Vim告知未被显示的行数，行编号则用于确认并强调此项信息。

另请注意行编号左侧的灰色留白，这些位置保留下来，以供更多折叠提示线索的呈现。随着折叠的创建及使用，将看到Vim安插至这些位置的提示线索。

请注意图14-2中的光标位于第18行。我们输入zf2j，把第18行及其下两行放入一个折叠里。图14-3即为呈现效果。

```
  13
  14   if (thiscode == anysense)
  15     {
  16
  17     printf ("07 some other line\n");
+ 18 +--  3 lines: printf ("08 some other line\n");-----
  21
  22     }
  23
  24   printf ("07 some line\n");
```

图14-3：在第18行折叠3行

请注意Vim如何以+--作为前缀创建一个容易识别的标记，以及它如何列出折叠内容中第一行的文本，用于占位。再看屏幕最左侧，Vim加上一个+号，这也是视觉上的线索。

对同样一份文件，接下来要折叠if语句后、花括号间的代码块（也包括花括号所在的行）。把光标放在（开头或结尾）的花括号上，而后输入zf%。文件将做类似图14-4所示的变化。

```
    8   printf ("02 some line\n");
    9   printf ("03 some line\n");
   10   printf ("04 some line\n");
   11   printf ("05 some line\n");
   12   printf ("06 some line\n");
   13
   14   if (thiscode == anysense)
+  15 +--  8 lines: {--------------------------
   23
   24   printf ("07 some line\n");
   25   printf ("08 some line\n");
   26   printf ("09 some line\n");
```

图14-4：在if语句后折叠的代码块

这次共折叠了8行代码，其中有3行包含在前次创建的折叠中。这称为嵌套折叠（nested fold）。请注意嵌套折叠没有特殊标示。

我们的下一个实验是把光标移到第25行并往上折叠内容，直到（并包括）定义了fcn的地方。这一次我们使用Vim的搜索动作。折叠命令以zf开始，使用?int fcn（Vim的向后搜索命令）寻找到fcn函数的开头处，然后按下ENTER键。屏幕画面应如图14-5所示。

```
    3
    4
+   5 +-- 21 lines: int fcn (int v1, int v2)--------
   26   printf ("09 some line\n");
   27   printf ("10 some line\n");
   28   printf ("11 some line\n");
   29   printf ("12 some line\n");
   30
   31   }
~
```

图14-5：折叠至函数的开头处

注意： 如果你计算了行数并创建了一个跨越其他折叠的折叠（例如3zf），所有包含在被跨越的折叠里的内容都会被计算为一行。假设光标位于第30行，下面的第31行到第35行被隐藏在一个折叠里，则再下一行显示的为第36行，3zf创建了一个包含屏幕上3行的折叠：第30 行、折叠成一行的第31—35行以及接下来的第36行。听得一头雾水吗？有一点。我们可以这么想：zf 命令的行数计算规则就是“只计算看得见的行”。

让我们尝试其他功能。首先，使用zo命令（z后接字母o，不是数字0）打开所有折叠。现在看到一些关于折叠的线索出现在左边界处，如图14-6所示。这个边界里的每一栏都称为折叠栏（foldcolumn）。

```
    4
-   5 int fcn (int v1, int v2)
|   6   {
|   7
|   8   printf ("02 some line\n");
|   9   printf ("03 some line\n");
|  10   printf ("04 some line\n");
|  11   printf ("05 some line\n");
|  12   printf ("06 some line\n");
|  13
|  14   if (thiscode == anysense)
|- 15     {
||  16
||  17     printf ("07 some other line\n");
||- 18     printf ("08 some other line\n");
||| 19     printf ("09 some other line\n");
||| 20     printf ("10 some other line\n");
||  21
||  22     }
|   23
|   24   printf ("07 some line\n");
|   25   printf ("08 some line\n");
```

图14-6：所有折叠被打开

图中，每个折叠的第一行都附有减号（-），其他被折叠的行则标上竖线（|）。代表最大（外层）折叠的符号位于最左栏，最内层折叠的符号则位于最右栏。如图所示，第5行到第 25 行显然处于最外层的折叠（本例为第一层gvim），第15行到第 22行显然为向内一层的折叠（第二层），第18行到第20行则显示为最内层的折叠。

注意： 默认情况下，这么好用的可视线索却被设置为关闭（我们不知道原因，或许在于占用了屏幕画面的空间）。下列命令可打开这个功能，并设置它占用的宽度：

```
:set foldcolumns=n
```

其中，*n*是可视折叠线索使用的列数（最大为12，默认为0）。上图中，我们采用了foldcolumn=5（如果你注意观察，没错，较前面的几张图把foldcolumn设置为3，我们改变了设置值，以求较好的视觉展现）。

现在继续创建更多的折叠，观察它们的影响。

首先，重新折起最内层的折叠（涵盖第18行到第20行），把光标移到折叠范围内的任一行，输入zc（关闭折叠）。图14-7显示了结果。

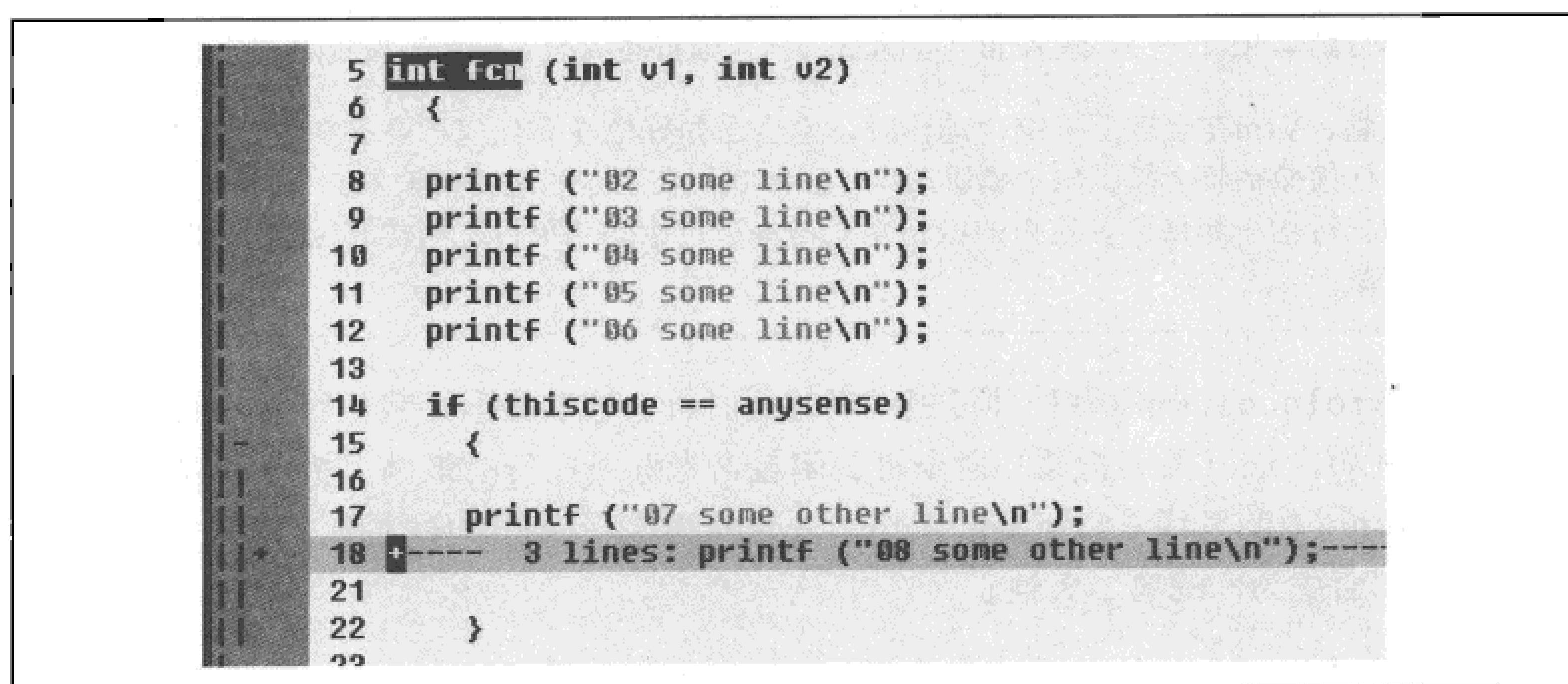


图14-7：重新折叠第18—20行之后

看到灰色边界的改变了吗？Vim也负责维护可视线索，让折叠的视觉表现与管理都很简单。

现在看看典型的"单行"命令对摺叠的影响。把光标放到折叠的行（18）。输入 ~~（为当前这一行的所有字符更改大小写）。请记住，在Vim中的~是个对象运算符（除非设置了compatible选项）。下一步，输入zo打开折叠，折叠里的代码现在将如图14-8所示。

这是个很方便的功能。对一行下达的命令或运算符能套用在折叠行代表的所有文本上！这个范例或许不太自然，但它恰当地勾画出了这项技巧的潜力。

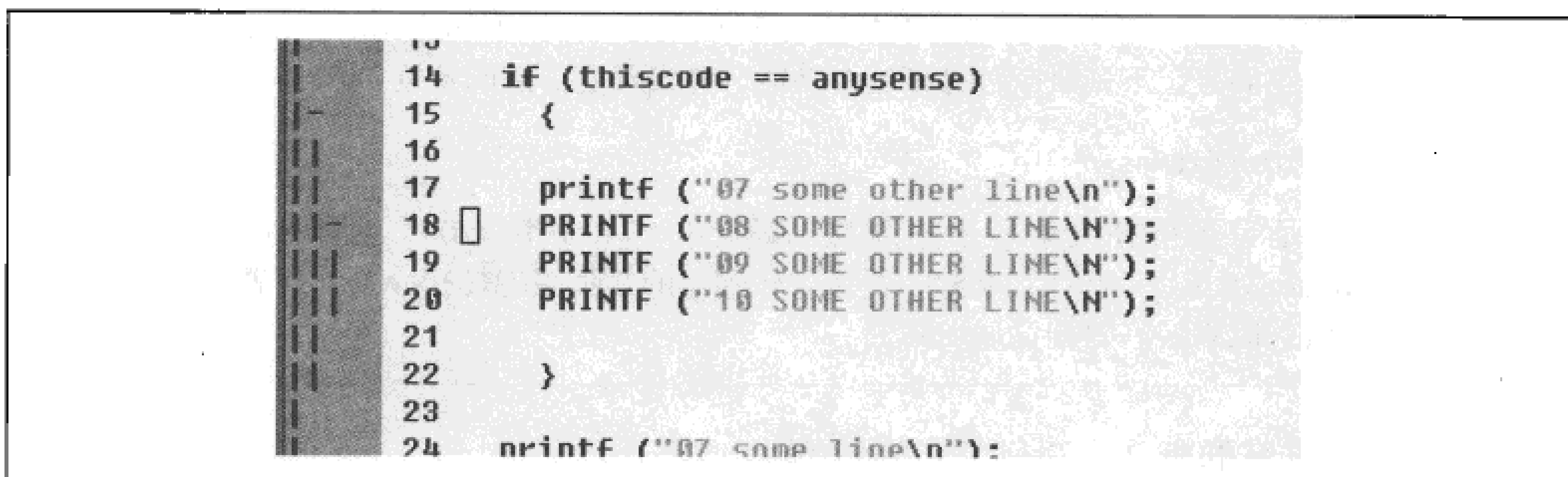


图14-8：大小写改变运用到一个折叠

注意： 任何对折叠采取的行动均影响到整个折叠。例如，把光标放在图14-7的第18行——隐藏了第18行到第20行——并输入dd（删除整行），则折叠内的三行都会被删除，并删除折叠。

请注意，Vim在管理所有编辑动作时，它会当作好像没有任何折叠存在（这点很重要），所以任何撤销将会撤销整个编辑动作。假如我们在刚才的改变后输入u（撤销），则刚才删除的三行均一起恢复。这里的撤销功能与本节讨论的“单行”动作有区别，但有时似乎又很相似。

现在是熟悉foldcolumn边界种种可视线索的好时机。这些线索让我们容易知道正在处理的折叠。例如，zc（关闭折叠）命令可关闭包含光标所在行的最内层折叠。我们可以通过foldcolumn中的竖线看出折叠的大小。一旦熟悉这些线索，像打开、关闭、删除折叠等行为，都会成为你的第二天性。

大纲

请看下列使用tab做缩排的简单（且刻意制作的）文件：

```
1. This is Headline ONE with NO indentation and NO fold level.
    1.1 This is sub-headline ONE under headline ONE
        This is a paragraph under the headline. Its fold
        level is 2.
    1.2 This is sub-headline TWO under headline ONE.
2. This is Headline TWO. No indentation, so no folds!
    2.1 This is sub-headline ONE under headline TWO.
       Like the indented paragraph above, this has fold level 2.
           - Here is a bullet at fold level 3.
                paragraph at fold level 4.
           - Here is the next bullet, again back at fold level 3.
       And, another set of bullets:
           - Bullet one.
           - Bullet two.
    2.2 This is heading TWO under Headline TWO.
3. This is Headline THREE.
```

可以使用Vim折叠形成伪大纲以观察你的文件。请把你的折叠方式定义为indent：

```
:set foldmethod=indent
```

在我们的文件中，定义shiftwidth（tab的缩排层次）为 4。现在即可根据各行的缩排而打开或关闭折叠。每个shiftwidth（本例中为4栏）遇到有缩排的行，该行折叠层次即递增1。例如，范例文件中的副标题行的缩排为一个shiftwidth，也就是4栏，因此折叠层次为1；缩排8栏（两个shiftwidth）的行，其折叠层次即为2……依此类推。

使用foldlevel命令可以控制可见的折叠层次。它接受一个整数参数，只显示折叠层次小于等于参数的行的内容。以我们的文件为例，下列命令要求只查看最高层折叠的标题：

```
:set foldlevel=0
```

屏幕画面将如图14-9所示。

```
1. This is Headline ONE with NO indentation and NO fold level.
+--  4 lines: 1.1 This is sub-headline ONE under headline ONE---------
2. This is Headline TWO.  No indentation, so no folds!
+--  5 lines: 2.1 This is sub-headline ONE under headline TWO.--------
+--  4 lines: And, another set of bullets:----------------------------
3. This is Headline THREE.
~
```

图14-9：折叠层次=0

设置foldlevel为 2，则可显示要点符号（-）以上层次的一切内容，如图14-10所示。

```
1. This is Headline ONE with NO indentation and NO fold level.
   1.1 This is sub-headline ONE under headline ONE
       This is a paragraph under the headline.  Its fold
       level is 2.
   1.2 This is sub-headline TWO under headline ONE.
2. This is Headline TWO.  No indentation, so no folds!
   2.1 This is sub-headline ONE under headline TWO.
       Like the indented paragraph above, this has fold level 2.
+----  3 lines: - Here is a bullet at fold level 3.----------------
       And, another set of bullets:
+----  2 lines: - Bullet one.--------------------------------------
   2.2 This is heading TWO under Headline TWO.
3. This is Headline THREE
```

图14-10：折叠层次=2

可利用这项技巧检阅你的文件，使用Vim的折叠递增（zr）或递减（zm）命令，即可快速地展开或折叠可见的层次细节。

其他折叠方式

我们的篇幅不够讨论所有折叠方式，但为了引起各位的兴趣，欢迎大家很快地翻阅*syntax*（语法）折叠方式。

借用稍早用过的C文件，但这一次我们让Vim根据C语法决定折叠的对象。掌握在C里折叠的规则很复杂，但我们的简单程序片段，已足以展示Vim的自动化能力。

首先，输入zD（删除所有折叠），确认清除所有折叠。画面上列出所有的代码，在foldcolumn的留白上没有任何可视标记。

采用下列命令，确认折叠功能已打开：

```
:set foldenable
```

（手动折叠前不需多此一举，因为foldenable于foldmethod设置为manual时即已自动设置。）接着输入如下命令：

```
:set foldmethod=syntax
```

折叠效果如图14-11所示。

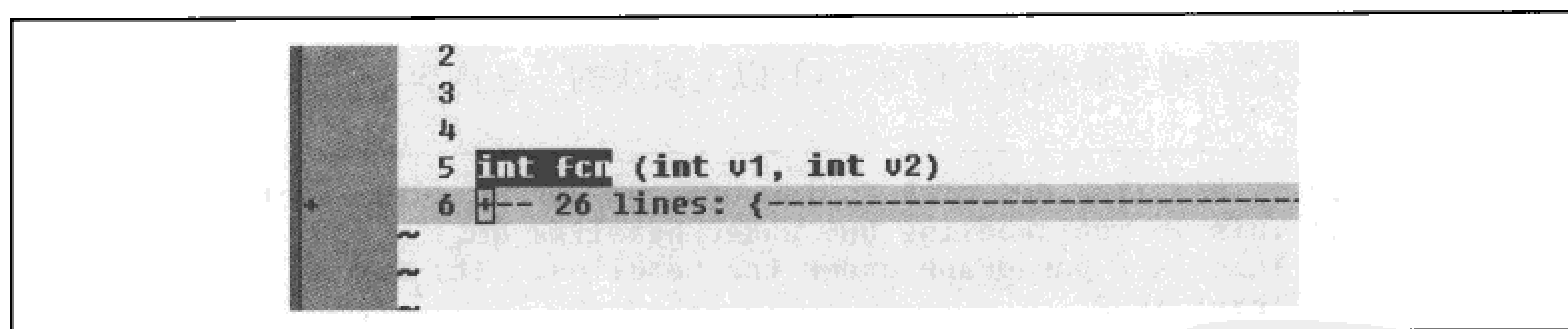


图14-11：在命令设置foldmethod=syntax之后

Vim折起所有花括号括起的代码块，因为这些块是C的逻辑语义块（logical semantic block）。如果你把光标移到本例的第6行，输入zo，Vim将展开这个折叠，并显示它的内部折叠层次。

每种折叠方式都使用不同的折叠定义规则。我们鼓励大家卷起自己的袖子，到Vim的说明文档里多多了解这些威力强大的方式。

Vim的diff模式（通过vimdiff命令调用）是个能与折叠、窗口、语法高亮显示等功能结

合的强力工具，稍后另有讨论。如图14-12所示，diff模式会显示文件间的差异，通常用于比较同一份文件的不同版本。

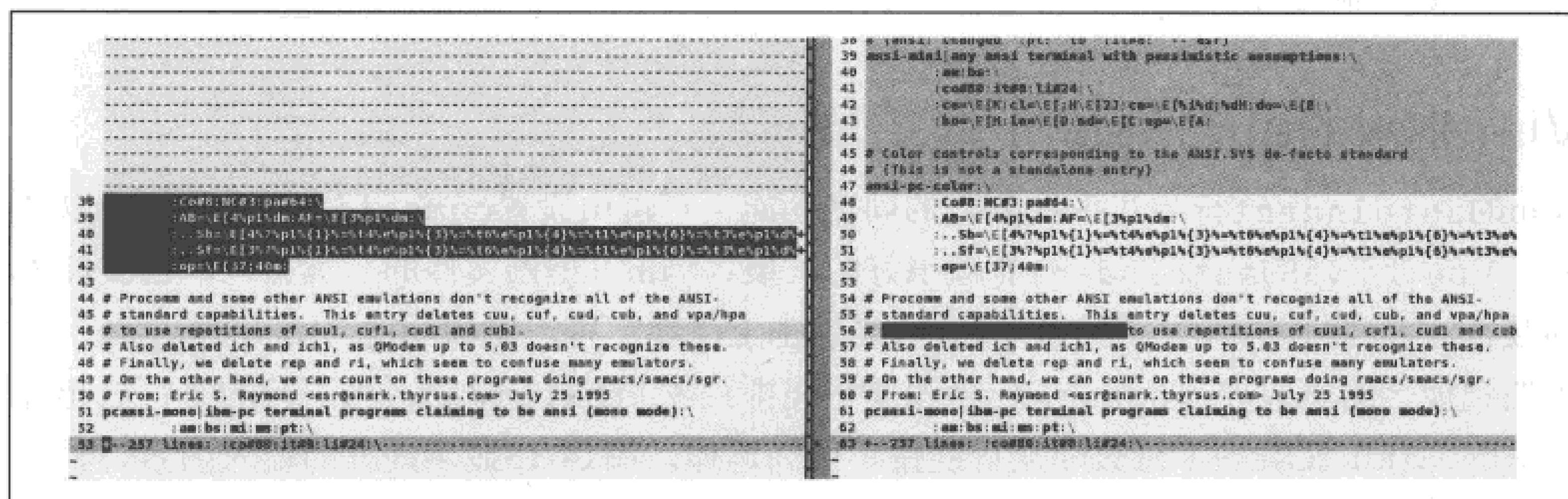


图14-12：Vim的diff功能及其折叠的使用

自动智慧缩排

Vim为自动缩排文本提供了四种复杂度与威力都递增的方式。其中最简单的形式，其运行方式几乎与vi的autoindent选项一模一样，而且Vim也使用相同的名称来描述这项行为。

只需在:set命令里指定即可选用所需的缩排方式：

```
:set cindent
```

Vim提供下列缩排方式（依其复杂度排列）：

autoindent
: 自动缩排方式模仿vi的autoindent选项。两者的细微差异仅在于缩排删除后的光标位置。

smartindent
: 比autoindent稍微强大一点，能识别基础C语法元素，用于定义缩排层次。

cindent
: 如其名所示，cindent嵌入了更多C语法理解能力，并在简单的缩排层次上引入了复杂的自定义方式。例如，cindent的配置能调整以配合你（或你上司）的编码风格，包括（但不只是）花括号（{}）如何缩排、花括号的位置安排、开头与结尾括号是否也要缩排……甚至能设置缩排如何查找内含的文本。

indentexpr
: 能让我们自定义表达式，Vim则根据开始每个新行时的上下文评估表达式。有了这

项功能，就能写出自己的规则。细节讨论请大家参考关于脚本编程与函数的讨论以及Vim说明文档。如果其他三种方式不能给你足够的自动缩排灵活性，indentexpr一定能满足你的需求。

Vim 对vi的autoindent 的扩展

Vim的autoindent行为几乎就像vi的同名功能，可借由设置compatible选项而予以区分。Vim对于vi的autoindent有项很好的扩展：Vim可识别文件的“类型”，当文件中的某行注释绕排到新的一行时，它会适当地加上注释字符。这项功能与wrapmargin（文本遇到规定的右边界wrapmargin栏即换行绕排）或 textwidth（当一行的字符数超过 textwidth 设置的字符数即换行绕排）协同运作。图14-13显示同样的输入内容，其一使用 Vim 的autoindent，另一种则使用vi。另外，在设置compatible选项后（用于模仿vi的行为），textwidth选项将失效，文本只会因wrapmargin的值绕排。

smartindent

smartindent对autoindent稍微做了一些扩展。它很好用，但如果代码采用类似 C 设计的编程语言，而且语法很复杂，最好省下你的时间，改用cindent。

smartindent自动插入缩排的时机，包括：

- 接在{后的下一行。
- 以cinwords选项中包含的关键字开始的行。
- 如果光标位于包含右花括号（}）的行且用户使用O命令（在当前位置的上方打开新行）创建新的一行，则新的一行创建在以}开始的行前。
- 内容以右花括号（}）结尾的新行。

cindent

经常利用Vim又采用类似C的语言做编程的用户，将会使用cindent或indentexpr。虽然indentexpr威力较强大、较有灵活性、更适合自定义，但cindent对大多数编程任务而言更为实用。它的许多设置已满足大多数程序员的需要（与共同标准）。请试着使用cindent的默认设置，如果与你的标准不同，再自定义缩排格式也不迟。

注意： 若是在indentexpr中已定义值，将覆盖cindent的动作。

---

cindent的动作通过三个选项定义：

cinkeys

定义让Vim重新估算缩排的键盘按键组合。

cinoptions

定义缩排样式（style）。

cinwords

定义标志着Vim应于后续内容行中加入额外缩排的关键字。

cindent使用cinkeys定义的字符串作为如何缩排的定义规则组合。我们将查看cinkeys的默认值，再讨论其他可以定义的设置以及这些设置的运作方式。

cinkeys选项

cinkeys是以逗号分隔的一些值：

```
0{,0},0),:,0#,!^X^F,o,0,e
```

以下列出各个值，依其上下文不同分隔，并简短说明其行为：

0{

0（零）为后面的{字符设置一行开头（beginning of line）的上下文。也就是说，如果一行中输入的第一个字符为{，Vim将重新估算该行的缩排。

别把这个选项里的0误以为“使用零缩排”——C语言缩排的常见习惯。这里的 0表示“如果字符输入的位置是一行的开头处”，而非强迫字符显示在一行的开头处。

{默认的缩排距离为零：除了现有的缩排层次，不另行增加缩排。下例显示了典型效果：

```
main ()
{
    if ( argv[0] == (char *)NULL )
    { ...
```

0}, 0)

如前一个选项的描述，这两个选项设置一行开头的上下文。因此，若于行开头输入} 或)，Vim即重新估算缩排。

}默认的缩排距离与成对的开始括号{定义的缩排距离相同。

)的缩排距离则是一个shiftwidth。

:

这是C的标签（label）或case statement上下文。如果:（冒号）是标签或case

statement语句的结尾，Vim即重新估算缩排。

:的默认缩排距离为1，即该行中的第一栏。别与零缩排弄混了，零缩排让新行与前一行维持相同的缩排层次。而当缩排距离为1，新行的第一个字符将向左移到第一栏。

0#

这也是一行开头的上下文。当#是某行中输入的第一个字符时，Vim即重新估算缩排。

默认缩排距离与前一个定义中的相同，也就是把整行移到第一栏。这点与第一栏里的开始宏（#define）的习惯一致。

!^F

特殊字符!定义后续的字符为重新估算当前行中的缩排的触发器（trigger）。本例的触发字符为^F，代表CTRL-F，所以默认行为是让Vim在我们按下CTRL-F 时重新估算当前所在行的缩排。

o

这个上下文定义我们创建的任何新行，无论是于插入模式中按ENTER键，或是使用o命令（打开新行）。

O

这个上下文涵盖使用O命令在当前的文本行之上(above)新行的创建。

e

这是*else*情况。如果某行以else初始化，Vim即重新估算缩排。直到else的最后一个“e”输入前，Vim都不会识别这种情况。

cinkeys syntax rules（cinkeys语法规则）。每个cinkeys定义的组成包括一个可选前缀（!、*或0）以及重新估算缩排的按键。各前缀意义如下：

!

指示让Vim重新估算当前行缩排的按键（默认为CTRL-F）。我们可以增加额外按键定义作为命令（使用+=语法），而不覆盖既有命令。换句话说，我们可以提供多组触发行缩排的按键。任何加入！定义中的按键仍可继续执行它的原有功能。

*

要求 Vim在插入指定按键前重新估算当前行的缩排。

0

设置一行开头的上下文。指定在0后的按键只在输入一行的第一个字符时才触发缩排的重新估算。

注意：请小心vi与Vim对“一行的第一个字符”与“一行的第一栏”的区别。各位已经知道，按下^可移动至一行的第一个字符，但不见得是第一栏（最靠左端的位置）；使用I插入内容时也是如此。同样地，前缀0应用于输入第一行的第一个字符，无论它是否靠左。

cinkeys具有特殊的关键字名称，而且提供覆盖任何保留字符（例如前述的前缀字符）的功能。专用的关键字选项如下：

<>

使用此形式照字面地定义关键字。对于特殊的非打印键，则使用它的念法。例如，字面量字符“:”可定义为<:>，非打印键“向上箭头”则可定义为<Up>。

^

使用^代表控制字符。例如^F定义了按键组合CTRL-F。

o、O、e、:

我们在cinkeys的默认值里看到过这些值。

=*word*、=~*word*

使用这些定义接受特殊行为的词汇。一旦出现匹配的字符串*word*，而且是新行的第一个文本，Vim即重新估算缩排。

=~*word*的形式与=*word*完全一样，但它忽略大小写差异。

注意：术语*word*（词）是个不幸的错误。更恰当地说，它代表词的开头，因为只要字符串匹配就会触发行动，但这并未要求字符串结尾必须也是词的结尾。在Vim的内置说明文档中，讲到end可匹配出end与endif。

cinwords选项

cinwords定义关键字，输入这些关键字时，将在下一行触发额外的缩排。此选项的默认值是：

```
if,else,while,do,for,switch
```

已涵盖C语言的标准关键字。

注意：这些关键字有大小写之分，Vim甚至无视ignorecase选项的设置。如果你需要的关键字的大小写有多种组合，则需于cinwords字符串里指定所有组合。

cinoptions选项

cinoptions控制Vim在C的上下文中重新缩排文本行的行为，其中包括控制一些代码格式化标准的设置，例如：

- 由花括号（{}）括起的代码块的缩排距离
- 当花括号接在条件语句后时是否插入新的行
- 如何根据括起代码块的花括号而对齐

cinoptions以其默认值定义了28种设置：

```
s,e0,n0,f0,{0,}0,^0,:s,=s,l0,b0,gs,hs,ps,ts,is,+s,c3,C0,/0,(2s,us,U0,w0,W0,
    m0,j0,)20,*30
```

这个选项值的长度应该能让各位体会到Vim自定义缩排的方式有多少种。大多数用cinoptions自定义的缩排依据块上下文而略有不同。有些自定义缩排定义扫描的距离（在文件里往前与往后的距离），以创建正确的上下文并恰当地估算缩排。

根据各种上下文而更改缩排距离的设置，可以递增或递减缩排的层次。另外，也可以重新定义缩排使用的列数。例如，设置cinoptions=f5能使开始花括号（{）的缩排为5列，只要它并非位于其他花括号内。

另一种定义缩排增量的方式，是使用某些shiftwidth的倍乘数（不必为整数）。以前例而言，把w附加到定义中（即cinoptions=f5w），开始花括号即移动5个shiftwidth的距离。

在任何数值前加入减号（-），可向左改变缩排层次（表示负向缩排）。

警告： 这个选项及其字符串值在调整时需要非常小心。还记得使用= syntax即可完全重新定义一个选项吗？因为cinoptions可能携带太多设置，请使用只调整极小部分的命令做改变：+=用于增加设置，-=删除既有设置，-=后接+=则可改变现有设置。

接下来简短列出各位最可能想修改的选项。这只是cinoptions设置中的极小部分子集，许多读者或许将发现其他（甚至全都是）适合自定义的设置。

>n（默认为s）

需要缩排的任何行应该缩排的位置为*n*。默认值为s，表示默认的一行缩排距离为一个shiftwidth。

fn, {n

f定义一个非嵌套的开始花括号（{）的缩排距离。默认值为零，让花括号与它们在逻辑上的另一半括号对齐。例如，一个接在while行下的花括号将被放置在while的“w”下。

{与f的行为相同，但套用至嵌套的开始花括号。它的默认缩排层次也是零。

图14-14与14-15显示了两个在Vim里缩排文本的范例，第一个范例的设置为cinoptions=s,f0,{0，第二个范例的设置为cinoptions=s,fs,{s。这两个范例的shiftwidth均为4（4列）。

```
18
19 while (condition)
20 {
21     if (someothercondition)
22     {
23         printf("looks like I've got both conditions!\n");
24     }
25 }
26
```

图14-14：cinoptions=s,f0,{0

```
27 while (condition)
28     {
29     if (someothercondition)
30         {
31         printf("looks like I've got both conditions!\n");
32         }
33     }
34
```

图14-15：cinoptions=s,fs,{s

}*n*

使用这项设置定义结尾花括号（}）的偏移量（与相对应的花括号而言）。默认值为零（对齐对应的大括号）。

^*n*

如果开始花括号位于第一列，则在一对花括号间（{}）的内容的当前缩排距离增加*n*。

:*n*, =*n*, b*n*

这三个设置控制了case语句的缩排。使用:*n*，Vim将case的标签缩排*n*个字符，从对应的switch语句开始计算。默认值为一个shiftwidth。

=设置根据相对应的case标签定义各行的缩排。这些语句的默认缩排值为一个shiftwidth。

b设置定义放置break语句的位置。默认值（零）让break与相对应case块中的其他语句对齐。任何不为零的值均让break与相对应的case标签对齐。

*)n, *n*

这两项设置扫描行数，分别让Vim寻找非结尾括号（默认为20行）以及未结束的注释（默认为30行）。

注意： 显然，这两个设置限制了Vim查找匹配文本的努力。有了时下威力强大的计算机后，各位可考虑向上调整这些值，以确保查找注释与括号时有更完整的作用域管理。试着扩大扫描范围为默认值的两倍，分别从40与60开始尝试。

---

indentexpr

如果定义了indentexpr，它将覆盖cindent，使我们自定义缩写规则，为编程语言编辑所需而量身订做。

indentexpr定义表达式，文件中每次创建新行时都要估算此表达式。这个表达式决定Vim用于新行缩排的整数。

除此之外，indentkeys能像cinkeys一样定义有用的关键字，出现这些关键字后，即重新估算该行的缩排。

但问题是，为一个语言从头开始设计自定义缩排规则，可不是简单的事情。幸好，这项工作多半已经完成了。请查找$VIMRUNTIME/indent目录，看看你惯用的语言是否已在其中。现在已有70多份缩排文件。

最常见的编程语言都在目录里，包括*ada*、*awk*、*docbook*（它的缩排文件称为docbk）、*eiffel*、*fortran*、*html*、*java*、*lisp*、*pascal*、*perl*、*php*、*python*、*ruby*、*scheme*、*sh*、*sql*、*zsh*，甚至还有为xinetd定义的缩排文件。

在.vimrc文件中加入filetype命令，我们可以要求Vim自动检测文件类型，并载入对应的缩排文件。现在，Vim将试着检测编辑中的文件类型，并载入相应的缩排文件。如果缩排规则不能满足你的需求——例如采用你不熟悉或不想要的风格，可使用:filetype indent off命令关闭缩排。

我们鼓励资深用户多多探索，并从Vim提供的缩排定义文件中学习。如果你发展出新的定义文件或改良了现有的文件，我们都鼓励大家上传到vim.org，有可能加入Vim包哦。

关于缩排的最后注意点

在结束我们的讨论前，值得再提醒一些使用自动缩排时的重点：

不做自动缩排时

任何时候，只要在编辑会话中“手动”对自动缩排的行改变它的缩排方式，Vim 将为该行加上标志，以后不会再试着对该行自动缩排。

复制与粘贴

当我们粘贴文本到文件中打开自动缩排的地方时，Vim把粘贴的文本视为一般输入而应用所有自动缩排规则。在多数情况中，自动缩排规则都不会符合我们的理想。任何对粘贴文本的缩排会附加在已应用的缩排规则。通常造成粘贴文本缩排太多而挤在画面右边，并未相应地向左边撤退。

为了避免奇怪的状况，并避免在粘贴文本时出现副作用，请在增加输入文本前设置Vim的paste选项。paste选项调整所有Vim的自动功能，以保证仅仅粘贴文本。想回到自动缩排模式时，只需用:set nopaste命令重设paste选项。

关键字与字典词汇补全

Vim提供了一套无所不包的“插入－补全”（insertion-completion）功能套件。从编程语言专用的关键字到文件名称、字典词汇，甚至是一整行文本，Vim知道如何为未完成的文本提供可能的补全选项。不只如此，Vim还抽取基于字典的补全选项的语义，使补全功能包括了基于已完成词汇的同义词。

本节中，我们将探讨不同的补全方式、这些方式的语法，并以范例说明各种补全方式的运作。补全方式包括：

- 整行补全
- 以当前文件关键字补全
- 以dictionary选项关键字补全
- 以thesaurus选项关键字补全
- 以当前及包含文件关键字补全
- 以标签补全（如同于ctags）
- 文件名称补全
- 以宏补全
- 以Vim命令行补全
- 依用户定义而补全
- 以omni补全

- 拼写建议补全
- 以complete选项关键字补全

除了complete关键字，所有与补全相关的命令均以CTRL-X开始。第二组按键则特别定义Vim要尝试的补全类型。例如，自动补全文件名称的命令是CTRL-X CTRL-F（可惜并非所有命令都这么易懂又好记）。Vim使用尚未映射的（默认的）键，让我们能减少多数命令到只需使用第二组按键（因为可以适当地映射到命令。例如，你可以把CTRL-X CTRL-N映射到CTRL-N）。

所有补全的方式几乎都有一样的行为：在我们输入第二组按键时，会循环重复一份候补补全内容名单。以通过CTRL-X CTRL-F 选择文件名称的自动补全为例，如果第一次没有出现正确的字，我们可以重复按下CTRL-F以查看其他选择。另外，如果按下CTRL-N（表示“下一个”），则会移动到下一种可能内容，按下CTRL-P（表示“前一个”）则向前移动。

让我们观察一些自动补全的方式及样例并思考它们的用处。

补全插入的命令

这些命令的功能范围从单纯地在当前文件中查找词汇，到函数、变量、宏以及整套代码中的其他名称。最终补全的方式要结合其他功能，在威力与复杂度间取得良好的平衡。

注意：各位可能想把自己惯用的补全方式找出来并映射至单一快捷键。我自己利用Tab键：

```
:imap Tab <C-P>
```

虽然牺牲了轻松插入tab字符的能力，但让我能使用在DOS、shell（如xterm、konsole）等命令行环境中的相同按键（默认按键），补全只输入了部分的信息（请记住，将tab的前后以CTRL-V括起就能插入tab）。将命令映射至Tab键与一般在Vim命令行模式中的映射键相同。

整行补全

通过调用CTRL-X CTRL-L补全整行。这种方式回溯当前的文件，寻找匹配已输入字符的行。我们试用一个范例让各位更清楚整行补全的运作。

试想象有份文件包含终端（terminal）或控制台（console）定义，定义中描绘了终端的功能及操纵方式。假设你的画面组成如图14-16所示。

请注意高亮显示的地方，包含“This terminal widely used in our company...”字

```
 1 #       Reconstructed via infocmp from file: /etc/terminfo/r/rxt
 2 # (untranslatable capabilities removed to fit entry within 1023
 3 # This terminal widely used in our company...
 4 rxvt-unicode|rxvt-unicode terminal (X Window System):\
 5         :am:bw:eo:hs:km:mi:ms:xn:xo:\
 6         :co#80:it#8:li#24:lm#0:\
 7         :AL=\E[%dL:DC=\E[%dP:DL=\E[%dM:DO=\E[%dB:IC=\E[%d@:\
 8         :K1=\EOw:K2=\EOu:K3=\EOy:K4=\EOq:K5=\EOs:LE=\E[%dD:\
 9         :RI=\E[%dC:SF=\E[%dS:SR=\E[%dT:UP=\E[%dA:ae=\E(B:al=\E[L
```

图14-16：按行补全的范例

样。你在很多地方都需要使用这行文字，正如文字内容所示，公司里到处都在使用这个终端。输入该行中的足够文字，让它尽量独一无二，然后按下CTRL-X CTRL-L。这时，图14-17包含部分输入文本行：

```
4 rxvt-unicode|rxvt-unicode terminal (X W
5         :am:bw:eo:hs:km:mi:ms:xn:xo:\
6         :co#80:it#8:li#24:lm#0:
7
8 # Thi
9 rxvt-unicode2|rxvt-unicode2 terminal (X
0         :AL=\E[%dL:DC=\E[%dP:DL=\E[%dM:I
```

图14-17：部分输入的行等待补全

```
# Thi
```

在CTRL-X CTRL-L的作用下，Vim列出可能的整行补全列表，列表根据稍早在文件中输入的内容列出。这份补全列表列于图14-18。

```
 8 # This terminal widely used in our company...
 9 # This section lists entries in a least-capable to most-cap
10 # This should only be used when the terminal emulator canno
11 # This entry is good for the 1.1.47 version of the Linux co
12 # This trick could work with other Intel consoles like the
13 # This terminal widely used in our company...
```

图14-18：键入CTRL-X CTRL-L之后

在灰度图像中这不容易看出来，但屏幕上提供了彩色的弹出窗口，窗口中包含匹配部分行内容的多条文本信息，这在屏幕截图上看不见，同时还显示了相匹配文本的位置信息。

这个方法使用complete选项来定义查找相匹配文本的范围。关于范围的确定，将在讨论下一个方法时详细说明。

随着我们在弹出的列表（注1）中向前移动（CTRL-N）或向后移动（CTRL-P），列表中高亮显示的选项也会改变。按下ENTER即可选择符合需求的选项。如果没有满意的选项，则按下CTRL-E，停止匹配且不替换任何文本。光标回到输入部分内容后的原始位置。

图14-19显示了从列表中选择任意选项的结果。

```
 3 # This terminal widely used in our company...
 4 rxvt-unicode|rxvt-unicode terminal (X Window System):\
 5         :am:bw:eo:hs:km:mi:ms:xn:xo:\
 6         :co#80:it#8:li#24:lm#0:
 7
 8 # This terminal widely used in our company...
 9 rxvt-unicode2|rxvt-unicode2 terminal (X Window System)2:\
10         :AL=\E[%dL:DC=\E[%dP:DL=\E[%dM:DO=\E[%dB:IC=\E[%d@
```

图14-19：键入CTRL-X CTRL-L并选择匹配行之后

以文件中的关键字补全

CTRL-X CTRL-N在当前的文件中向前搜索匹配光标前方字符的关键字。输入快捷键后，可使用CTRL-N或CTRL-P分别向前或向后搜索。按下ENTER即可选择想要的选项。

注意： 请注意此处的“关键字”采用宽松定义。它可以是程序员熟悉的“关键字”，也可以匹配出文件中的任意词汇（word）。词汇的定义为iskeyword选项中的任何连续字符组。iskeyword的默认值已很全，但如果想要纳入或排除某些标点符号，我们可以重新定义该选项。iskeyword中的字符可以直接指定（例如a—z）或以ASCII码指定（例如使用97—122代表a—z）。

举例来说，默认值允许下划线作为词汇的一部分，但把点号或连字符视为定界符。这项规则适用于类似C的程序语言，但或许不是其他环境中的最佳选择。

以dictionary补全

CTRL-X CTRL-K向前搜索在dictionary选项中定义的关键字，寻找是否有匹配光标前方字符的关键字。

dictionary选项的默认值未定义。在下述几个地方常常能找到dictionary文件，各位也可以自己动手定义。最常见的dictionary包括：

- /usr/dict/words（在XP上的Cygwin）
- /usr/share/dict/words（FreeBSD）

- $HOME/.mydict（个人的字典词汇）

请注意， Windows XP的字典词汇文件由 Cygwin（*http://www.cygwin.com/*）提供，Cygwin是 Unix 实用工具的自由软件仿真套件。虽然安装Cygwin已经超出本书的讨论范围，但还是提醒一下，安装时可以只选择安装其中的一小部分，而各位或许将发现，安装包含字典（dictionary）的部分将很有用处。

以thesaurus补全

CTRL-X CTRL-T向前搜索由`thesaurus`选项定义的文件，寻找是否有匹配光标前方字符的关键字。

这个方式提供了有趣的选项。当Vim找到相匹配文本后，如果在`thesaurus`文件中的那一行包含多个词汇，Vim则把所有列出的词汇放入候补列表里。

表面上（亦是选项的名称所暗示的），这种方式提供同义词，但允许我们定义自己的标准。以包含下列诸行的文件为例：

```
fun enjoyable desirable
funny hilarious lol rotfl lmao
retrieve getchar getcwd getdirentries getenv getgrent ...
```

前两行是典型的英语同义词（分别匹配“fun”与“funny”），但第三行则可能对常插入以get开始的函数名称的C程序员很有用。我们用于这些函数的同义词是“retrieve”。

实际上，我们会区分英语中的同义词与C语言中的同义词，因为Vim可以搜索多份同义词文件。

在输入模式中，输入`fun`，然后按下CTRL-X CTRL-T。图14-20显示了`gvim`中弹出的结果。

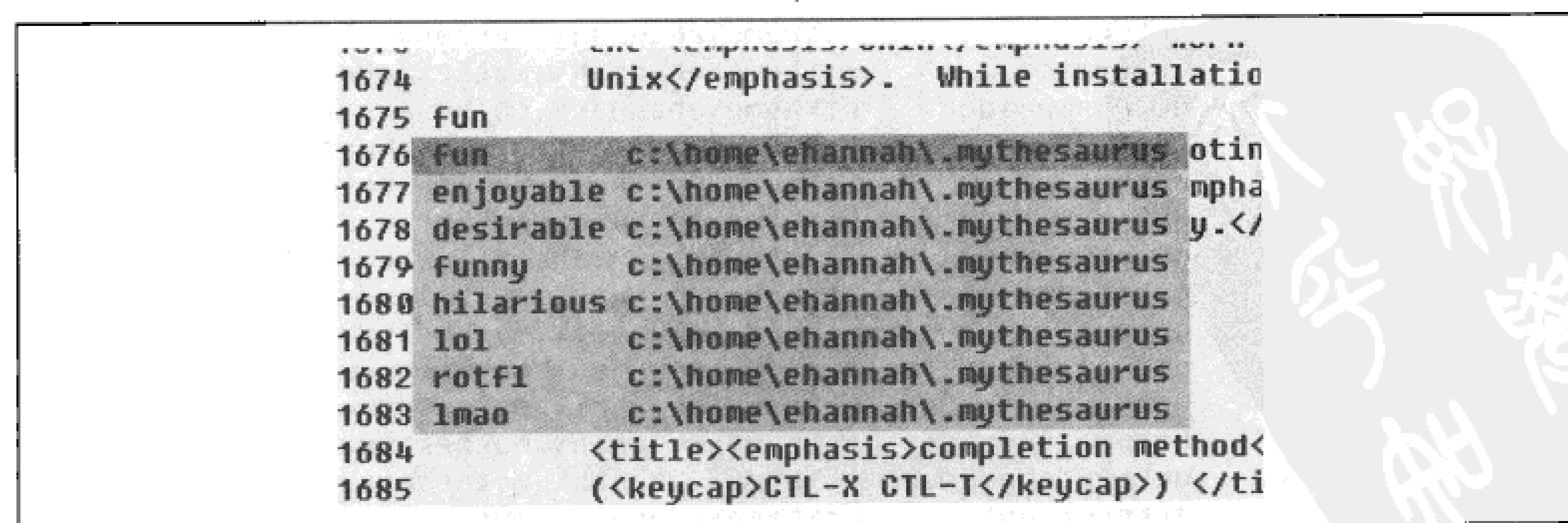


图14-20：“fun”的同义词补全

请注意：

- Vim匹配可在同义词记录中找到的任意词汇，而不只是同义词文件中的每一行的第一个词。
- Vim包含来自同义词文件中各行的候选词汇，这些词汇匹配光标前方的关键字。因此，本例将找出“fun”与“funny”。

注意：另一项有趣且或许意想不到的thesaurus行为，则是匹配能发生在同义词文件中一行上的任何地方，不只是第一个词。以前面的范例而言：

```
funny hilarious lol rotfl lmao
```

如果你输入了hilar并试着补全这个单词，Vim列出的列表中，将包含该行中hilarious以后的每个词汇，也就是包含“hilarious”、“lol”、“rotfl”、“lmao”。有趣吧！

有人注意到候选列表中的额外信息吗？把preview加入completeopt选项中，即可向Vim取得候选词汇的位置信息并显示在弹出菜单中。

再来看个范例，仍使用稍早用过的文件，输入部分词retrie。如此匹配“retrieve”便于推测以“get”开头的函数的同义词，然后把所有包含“get”的函数名称都列为同义词。现在，CTRL-X CTRL-T给我们一份弹出菜单（使用gvim时），菜单中列出所有函数，作为补全词汇的选项。请见图 14-21。

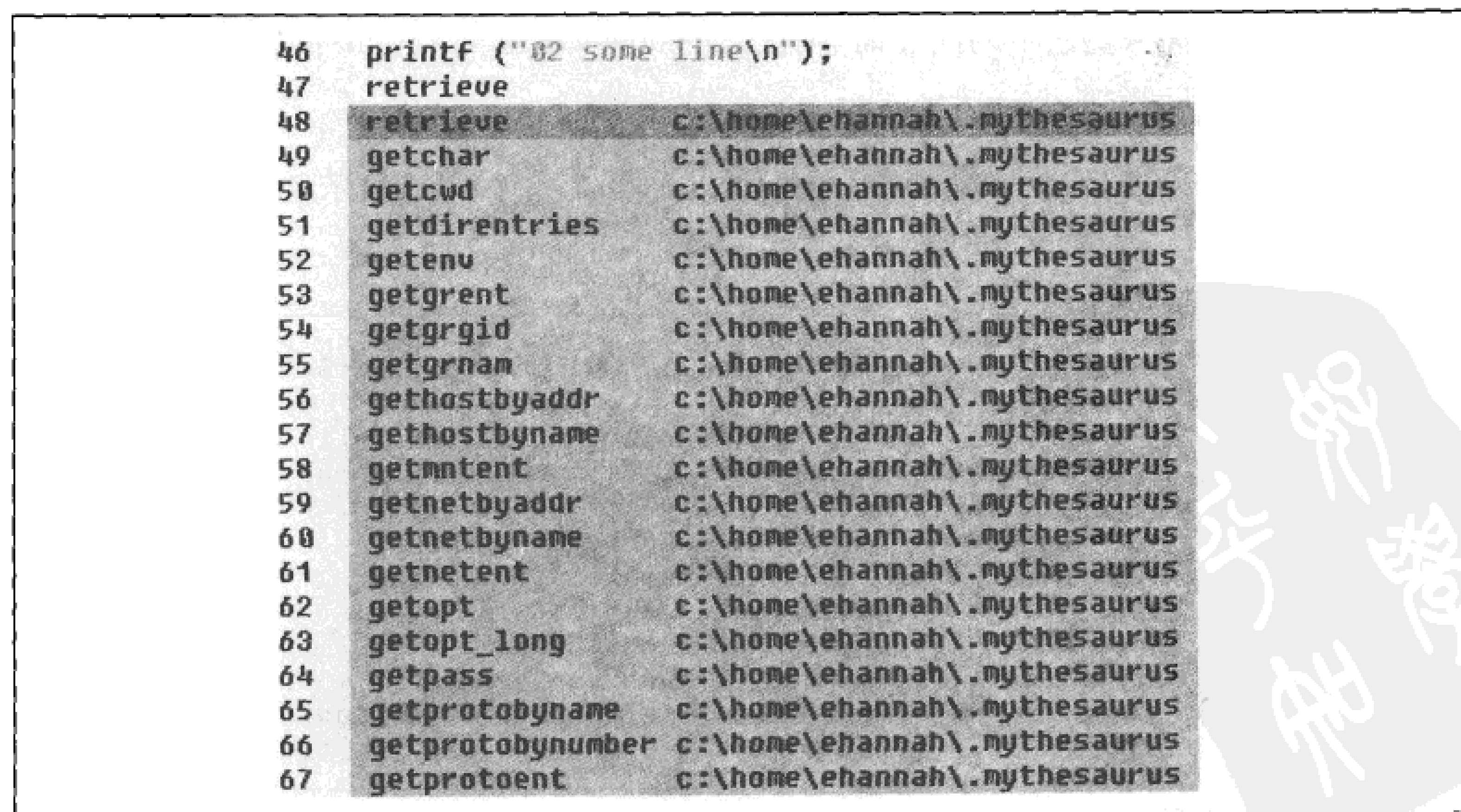


图14-21：字符串“retrie”的同义词补全

与其他完成方式一样，按下ENTER即可选择满意的选项。

以当前文件及包含文件中的关键字补全

CTRL-X CTRL-I向前搜索当前文件及包含文件（included file）中匹配光标前方字符的关键字。这个方法与“搜索当前文件”不同（CTRL-X CTRL-P），Vim检查当前文件对包含文件的引用并搜索这些包含文件。

Vim使用include值检测引用包含文件的行。默认值是一种模式，告诉Vim寻找匹配标准C结构的行：

```
# include <somefile.h>
```

本例中，Vim将至系统标准包含文件目录中的somefile.h文件中寻找。Vim也会使用path选项，作为搜索包含文件时的目录列表。

以标签补全

CTRL-X CTRL-J向前搜索当前标签与包含文件中匹配标签（tag）的关键字（关于标签的讨论，请参考第126页的“使用标签”）。

文件名的补全

CTRL-X CTRL-F搜索匹配光标前字符的文件名称。请注意，这个方式是让Vim以文件名称补全关键字，而不是以文件中找到的词汇补全。

注意： 到了 Vim 7.1，Vim在搜索可能的匹配文件名时，只搜索活动目录。这个行为与许多 Vim功能都使用path来寻找文件不一致。内置的Vim说明文档中暗示这个行为只是暂时性的，只是“尚未使用”path。

以宏与定义名称补全

CTRL-X CTRL-D向前搜索当前文件及包含文件中宏名称及#define指令所做的定义。这个方法检查当前文件，搜索包含文件的引用后一并搜索包含文件。

以Vim命令行补全

这个方法以CTRL-X CTRL-V调用，设置用于Vim命令行并试图猜测补全词汇的最佳选择。提供这种方法，在于协助用户开发Vim脚本。

以用户函数补全

这个方法以CTRL-X CTRL-U调用，让我们以自己的函数定义补全内容。Vim使用

completefunc选项指定的函数进行补全。关于脚本编码及Vim函数的编写，请参考第十二章。

以omni函数补全

这个方法以CTRL-X CTRL-O调用，使用自定义函数（很像前一种用户函数的方式）。两者显著的差异在于这个方法预期函数依文件类型而不同，因而在载入文件时即决定并载入所需函数。omni的补全文件已可支持C、CSS、HTML、JavaScript、PHP、Python、Ruby、SQL、XML。在内置的Vim说明文档中提到，Vim 7.1将提供更多脚本，包括 C++ 的omni函数文件，以鼓励大家试验这些文件的使用。

拼写建议补全

这个方式以CTRL-X CTRL-S调用。以光标前的字符作为基础，让Vim提供补全词汇的候选列表。如果词汇的拼写显然有误，Vim则会建议“较正确”的拼写方式。

以complete选项关键字补全

这是个最通用的选项，以CTRL-N调用，可让我们结合其他所有搜索。对许多用户而言，这应该是最为满意的方式，使用这个方式时，我们不需非常了解更专门的方式的微妙之处。

在complete选项中，设置可用来源列表（其间以逗号分隔），即定义了补全行动的地点及方式。每个可用来源用一个字符代表。选择包括：

. *（点号）*

搜索当前的缓冲区

w

搜索其他窗口中的缓冲区（但在包含Vim会话的屏幕内）

b

搜索缓冲区列表中其他已载入的缓冲区（可能不能在任何Vim窗口中看到）

u

搜索缓冲区列表中未载入的缓冲区

U

搜索不在缓冲区列表中的缓冲区

k

搜索字典文件（列在dictionary选项中）

kspell

使用当前的拼写检查方案（它是唯一超过一个字符的选项）

s

搜索同义词文件（列在thesaurus选项中）

i

搜索当前文件与包含文件

d

在当前文件与包含文件中搜索定义的宏

t、]

为了补全标签而搜索

关于Vim自动补全功能的最后注意点

我们给了许多关于自动补全的素材，但还有许多遗漏。投资在熟练自动补全上的时间将带来很大的回报。如果你经常需要编辑，而且有任意表示法（notion）或文本上下文需要补全，请试着找出最适合你的方式并好好学习。

最后一点。两组按键结合（如果你是个典型的Unix用户，把按键组合视为“不只一个键”）可能会引起错误，尤其是这里的按键组合都结合了CTRL键。如果觉得自己常会用到自动补全，可考虑把最常用的自动补全方式只映射到一个或一组按键。大量自动补全的命令都会缩短成只有原命令的一半，提高了不少效率。

为何我们发现这种自定义方式很有价值呢？稍早提过，我把Tab键映射到常用的关键字搜索。在使用DocBook XML标签编辑本书时，我输入“emphasis”超过1 200次（对书籍文件使用grep得到的结论）！使用了关键字补全，我知道“emph”只会匹配出一个选择，也就是我想要的“emphasis”标签。因此，每次出现这个词时，我至少省下按另外三个键的力气（假设输入前三个字母时都很成功），合起来总共省下3 600次的按键动作！

再提供另一个衡量完成效率的方法：我已经知道自己一秒大约输入4个字符，把输入这个关键字省下的3 600次按键除以4，表示省下了15分钟的时间。在同样一份 DocBook 文件中，我还另外设置了二、三十个类似的关键字，则省下的时间快速增加！

标签堆栈

第134页的“标签栈”一节提过相同主题。除了可以在搜索的标签间往复来回，还可以

在许多匹配搜索模式的标签中选择。我们也可以用一个命令完成标签选择与窗口分割。Vim里用于标签的ex模式命令已列于表14-1。

表14-1：Vim的标签命令

命令	函数
ta[g][!] [*tagstring*]	如tags文件中的定义，编辑包含*tagstring*的文件。如果当前的缓冲区已被修改但尚未保存，!可强迫Vim切换至新文件。 文件是否会被另外写入，根据autowrite选项的设置而定
[*count*]ta[g][!]	跳至标签栈中第*count*个新条目
[*count*]po[p][!]	把光标位置推出栈，恢复光标至它的原位置。如果有*count*，则移到第*count*旧条目
tags	显示标签栈的内容
ts[elect][!] [*tagstring*]	利用标签文件中的信息，列出匹配*tagstring*的标签。如果没有指定*tagstring*，则使用标签栈中最后一个标签的名称
sts[elect][!][*tagstring*]	与:tselect相似，但是为选择的标签分割窗口
[*count*]tn[ext][!]	向下跳到第*count*个匹配的标签（默认值为1）
[*count*]tp[revious][!]	向前跳到第*count*个匹配的标签（默认值为1）
[*count*]tN[ext][!]	
[*count*]tr[ewind][!]	跳到第一个匹配的标签。如果提供了*count*，则跳到第*count*个匹配的标签
tl[ast][!]	跳到上一个匹配的标签

正常来说，Vim会显示跳到了哪一个标签以及总共有几个标签。例如：

```
tag 1 of >3
```

大于符号（>）表示还没有尝试所有的匹配结果。你可以使用:tnext或:tlast来尝试更多的匹配。如果由于产生了别的消息而看不到这个消息，可以使用:0tn命令。

前述:tags命令的输出结果如下，当前的位置由大于符号（>）标示：

```
  # TO tag     FROM line in file
  1 1 main            1 harddisk2:text/vim/test
 >2 2 FuncA          58 -current-
  3 1 FuncC         357 harddisk2:text/vim/src/amiga.c
```

:tselect命令可以让你从多个匹配的标签中作选择。而“优先级”（pri字段）则指示

匹配的程度（全局或静态标签、是否能分辨大小写等等），这在Vim说明文档中有更详细的解说。

```
nr pri kind tag               file ~
 1F    f    mch_delay         os_amiga.c
                   mch_delay(msec, ignoreinput)
>2F    f    mch_delay         os_msdos.c
                   mch_delay(msec, ignoreinput)
 3F    f    mch_delay         os_UNIX.c
                   mch_delay(msec, ignoreinput)
Enter nr of choice (<CR> to abort):
```

:tag与:tselect命令可以接受一个以/开头的参数。此时，命令把参数当成正则表达式使用，Vim将搜索所有匹配正则表达式的标签。例如，:tag/normal可找出宏 NORMAL、函数normal_cmd等等。请使用:tselect /normal再加上所需标签的编号。

Vim命令模式的标签命令将于表14-2中介绍。除了可以像其他编辑器一样使用键盘，如果你的Vim启用了鼠标支持，也可以使用鼠标。

表14-2：Vim命令模式的标签命令

命令	功能
^] g <LeftMouse> CTRL-<LeftMouse>	寻找光标所在位置的标识符在tags文件中的位置，并移到那个位置。当前的位置会自动压入标签栈
^T	回到标签栈中的上一个位置，也就是弹出一个元素。前面加上数值，指定要弹出的元素数量

Vim中影响标签搜索的选项列于表14-3。

表14-3：Vim标签管理的相关选项

选项	功能
taglength, tl	控制要寻找的标签中有效字符的数量。默认值为零，表示所有字符都是有效字符
tags	其值是寻找标签所使用的目录与文件名列表。有一种特殊情况是，如果文件名以./开头，则点号会被替换成当前文件路径中的目录部分，让tags文件可以在不同的目录中使用。默认值是"./tags,tags"
tagrelative	如果被设为true（默认值）并且在其他目录中使用了tags文件，则tags文件中的文件名会被当成相对于tags文件所在的目录

Vim可以使用 Emacs样式的etags文件，这只是为了保证向下兼容性。这种格式在Vim的说明文档中没有介绍，也不鼓励使用etags文件。

最后，Vim也会寻找包含光标的整个单词，而不是只寻找从光标位置开始的一部分单词。

语法高亮显示

Vim对vi最有力的强化之一就是语法高亮显示（syntax highlighting）。Vim的语法格式极为依赖色彩的使用，但在不支持色彩的屏幕上也做了相当大程度的努力。本节将讨论三个主题：入门、自定义、从头开始动手做。Vim语法高亮显示包含的功能已超出本书范围，所以我们把重点放在提供信息，以便大家熟悉它的操作，并让各位能根据自己的需求扩展它。

注意： 因为Vim的语法高亮显示主要带来色彩层面的影响，但本书不是（彩色的），所以我们非常鼓励大家动手尝试语法高亮显示，这样才能全面体验色彩在定义上下文中的威力。我还没遇过使用过语法高亮显示后仍然拒用这项功能的用户。

入门

要显示一个文件的语法高亮显示很简单,只需执行如下命令：

```
:syntax enable
```

如果一切顺利，而且你也用正式的语法编辑文件，例如某种程序语言，应该会看到文本以各种颜色显示，这一切都由上下文与语法决定。如果什么都没改变，可试着打开语法设置：

```
:syntax on
```

一般打开语法设置应该就已足够，但我们遇过需要额外命令才能打开语法高亮显示的状况。

如果还是看不到语法高亮显示，可能是Vim不能识别你的文件类型，因此不了解使用哪种语法才合适。有几种原因会造成这个状况。

例如，你创建一个新文件且未使用Vim能识别的后缀（扩展名），或根本没有后缀，Vim则不能判断文件类型，因为该文件是新的并是空的。假如我编写了一个没加上.sh后缀的shell脚本。每份新的shell脚本的编辑生命都是从没有语法高亮显示开始。在文件中

逐渐加入代码后，Vim就可识别文件类型，语法高亮显示也将如预期般运作。

但Vim还是有可能（虽然可能性已经很低）不能识别你的文件类型。这种情况很少见，通常只需要明确地指定文件类型，因为已经有许多人为各种程序语言写好语法文件。而从头开始创建一份语法文件，是项复杂的终生事业,本章稍后会给大家一些提示。

从命令行手动设置语法可以强迫Vim使用我们选择的语法高亮显示方式。以编辑一份shell脚本为例，我总会如下定义syntax：

```
:set syntax=sh
```

第208页的“通过脚本动态配置文件类型”一节中，对于这个问题，展示了一个聪明又相当迂回的规避方式。

启用syntax后，Vim依照固定步骤设置语法高亮显示。以不陷入太多技术细节为前提，Vim 最终能决定我们的文件类型,找到适当的语法定义文件，并为我们载入该文件。语法文件的标准存储位置在$VIMRUNTIME/syntax目录里。

语法定义文件的范围究竟有多大？这么说吧，Vim的语法文件目录下包含将近500份语法文件。可使用的语法范围，从程序语言（C、Java、HTML）到内容（calendar），再到众所周知的配置文件（fstab、xinetd、crontab）等等。如果Vim不能识别你的文件类型，可试着在$VIMRUNTIME/syntax目录下寻找一个最接近的语法文件。

自定义

使用语法高亮显示后，各位或许发现有些颜色不适合你。可能是不太容易看出来，或者只是不符合你的审美观。Vim有几种自定义并调整颜色的方式。

在采取激烈手段前（例如动手编写自己的语法文件，如下一节所述），有几种方法可以尝试，以让语法高亮显示适合你的需求。

两项最常见也最具戏剧性的语法高亮显示不恰当行为为：

- 糟糕的对比，颜色太相近，很难看出彼此间的差异
- 太多、太花哨的色彩，使之看起来很恐怖

虽然这两项都是主观感受，但Vim能让我们动手改变，仍是很好的一件事。命令colorscheme与highlight以及选项background，大概能满足所有用户对色彩的偏爱。

还有几个其他命令与选项，能让我们自定义语法高亮显示。在简短介绍语法组后，我们将于后续章节讨论这些命令与选项，并特别重点介绍前面提出的三个命令与选项。

语法组

Vim把不同类型的文本分成组。这些组各自接收色彩与高亮显示定义。此外，Vim也允许组中的组。我们可以在不同层次着重不同定义。如果对包含子组的组指派定义，每个子组都会继承父组的定义，除非子组另有定义。

语法高亮显示时层次较高的组包含：

注释

程序语言专用的注释，例如：

```
// I am both a C++ and a JavaScript comment
```

常量

任何常量，例如TRUE

标识符

变量与函数名称

类型

声明，例如C语言里的int、struct

特殊

特殊字符，例如定界符

先不管“特殊”组，我们可以看看子组的范例：

- 特殊字符（SpecialChar）
- 标签
- 定界符
- 调试

对于语法高亮显示、组与子组有了基本认识后，就有了调整语法高亮显示所需的足够信息了。

colorscheme 命令

这个命令为不同语法高亮显示改变颜色，例如根据注释、关键字、字符串重新定义这些语法组。Vim发布时已附有下列配色方案选择：

- blue
- darkblue

- default
- delek
- desert
- elflord
- evening
- koehler
- morning
- murphy
- pablo
- peachpuff
- ron
- shine
- slate
- torte
- zellner

这些文件都在$VIMRUNTIME/colors目录中。使用如下命令即可启用想要的配色方案：

```
:colorscheme schemeName
```

注意：在非GUI的Vim里，可用另一种方法快速巡回采用不同的配色方案：只输入部分命令:color，再按下Tab键要求补全命令，然后按下空白键，最后重复按Tab键以巡回查看不同的配色方案选择。

在gvim中选择配色方案更为容易。单击Edit菜单，把鼠标移到Colorscheme子菜单，再选择“撕下”（有剪刀图示的虚线行）这个菜单。现在只要点击按钮，就能看到不同的配色方案选择了。

设置background选项

当Vim设置颜色时，它首先试着判断屏幕上的背景颜色。Vim只有两类背景色：深色或浅色。根据判断，Vim分别设置不同的颜色，希望与背景色搭配合适（搭配是指良好的色彩对比与兼容性）。虽然Vim非常努力，但正确地评估却非常困难，关于深色或浅色的评估很主观。有时候，对比配色把会话编排得刺目难看，有时候甚至不能阅读。

所以，如果颜色看起来不顺眼，可试着明确选择background设置。请先识别设置：

```
:set background?
```

这样你才知道设置是否改变。然后使用如下命令：

```
:set background=dark
```

使用background选项配合colorscheme命令，即可细致调整你的屏幕画面颜色。两者的结合通常能产生使人满意、看起来赏心悦目的配色。

highlight 命令

Vim的highlight命令能操纵不同组并控制编辑会话中不同组如何高亮显示。这个命令很强大。我们可以用列表的方式查看各种组的设置，也可以请求列出特定组的高亮显示信息。例如：

```
:highlight comment
```

这个命令用在我的编辑会话中，返回的结果如图14-22所示。

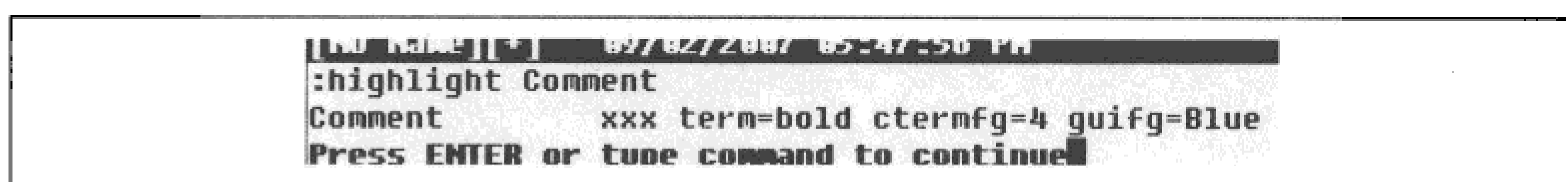


图14-22：注释的高亮显示

输出结果显示了在这个文件中的注解如何显示。本页上的xxx为深灰色，但在实际屏幕中，它是蓝色。term=bold表示在支持彩色的终端上，注释将以黑体（bold）呈现。ctermfg=4表示在彩色的终端上，例如彩色显示器上的xterm，注释的前景（文字）颜色将匹配DOS的深蓝色背景颜色。guifg=Blue表示图形用户界面显示的注释颜色，前景颜色将显示为蓝色。

注意： 与现在的图形用户界面相比，DOS能用的颜色组合具有较多限制。DOS中的颜色只有8种：black、red、green、yellow、blue、magenta、cyan、white。每一种都能设置为文本的前景色或背景色，亦可选择定义为“明亮”（bright），以在屏幕画面上显示较为明亮的色彩。Vim在非GUI的窗口中（如xterms）为定义文本颜色而使用类似映射。

GUI窗口提供了几乎无穷的颜色定义。Vim让我们可使用常见的名称定义颜色，例如Blue，但也一样能用RGB值定义颜色格式为#rrggbb：#为字面量，rr、gg、bb则是代表颜色层次的值。例如，鲜红色的定义为#ff0000。

使用highlight命令以改变你不喜欢的组设置。以图14-23为例，我们的 GUI 会把这个文件中的标识符显示为深青色。

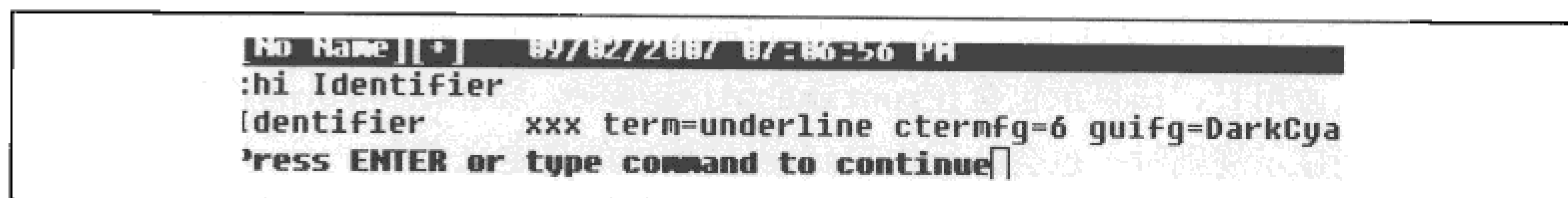


图14-23：标识符的高亮显示

```
:highlight identifier
```

下列命令可以重新定义标识符的颜色：

```
:highlight identifiers guifg=red
```

现在，所有画面上的标识符都变成（很丑的）红色了。这类自定义的配色非常没有灵活性：它会应用到各种文件上，而且不会因不同背景色或配色方案而改变。

若需要知道有多少高亮显示定义及值，一样可利用highlight：

```
:highlight
```

图14-24显示了highlight命令执行后的一小部分结果。

```
Constant        xxx term=underline ctermfg=1 guifg=Magenta
Special         xxx term=bold ctermfg=5 guifg=SlateBlue
Identifier      xxx term=underline ctermfg=6 guifg=DarkCyar
Statement       xxx term=bold ctermfg=3 gui=bold guifg=Brov
PreProc         xxx term=underline ctermfg=5 guifg=Purple
type            xxx term=underline ctermfg=2 gui=bold guifc
Underlined      xxx term=underline cterm=underline ctermfg=
Ignore              ctermfg=15 guifg=bg
Error           xxx term=reverse ctermfg=15 ctermbg=9 guifc
String          xxx links to Constant
Character       xxx links to Constant
Number          xxx links to Constant
Boolean         xxx links to Constant
Float           xxx links to Number
Function        xxx links to Identifier
```

图14-24：highlight命令的部分执行结果

请注意有几行包含有完整定义（列出了term、ctermfg……），同时有些行则接受父组的属性（例如String回溯到Constant）。

覆盖语法文件

在前一节中，我们学到如何一次定义整个语法组的属性。假设我们只想修改一两个语法

定义时，利用Vim的after目录即可完成需求。这是个可以创建无数*after*语法文件的目录，Vim将在执行正常语法文件后执行这些文件。

在after目录中的特定文件中可加入highlight命令（或任何处理命令——“after”处理的表示法可以通用），该目录包含在runtimepath选项中。现在，当Vim为你的文件类型设置语法高亮显示规则时，也会执行你在after文件中的自定义命令。

以应用自定义规则到XML文件为例。XML使用xml语法，也就是说，Vim从语法目录里的文件xml.vim载入语法定义。如前例所示，我们想把标识符定义为红色。所以先创建我们自己的xml.vim，存储在~/.vim/after/syntax目录下。在我们的xml.vim文件中，有这么一行：

```
highlight identifier ctermfg=red guifg=red
```

在执行上述自定义工作前，还需确保~/.vim/after/syntax已加入runtimepath路径：

```
:set runtimepath+=~/.vim/after/syntax      设置在.vimrc中
```

为了使改变永远固定，这一行当然要放入我们的.vimrc文件。

然后，当Vim载入xml的语法定义时，范例行将覆盖原始的标识符定义，改为我们定义的内容。

自己动手做

有了前一节打下的基础，自己设置一份语法文件的知识便已具备。就是这么简单，但在我们全面投入语法文件的开发前，还有很多方面需要学习。

我们将逐步创建一份语法文件。因为语法定义可能非常复杂，让我们先看一些简单、容易领会概念，但又复杂到能展现潜力的范例。

我们使用编造的Latin文件，loremipsum.latin：

```
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin eget
tellus. Suspendisse ac magna at elit pulvinar aliquam. Pellentesque
iaculis augue sit amet massa. Aliquam erat volutpat. Donec et dui at
massa aliquet molestie. Ut vel augue id tellus hendrerit porta. Quisque
condimentum tempor arcu. Aenean pretium suscipit felis. Curabitur semper
eleifend lectus. Praesent vitae sapien. Ut ornare tempus mauris. Quisque
ornare sapien congue tortor.

In dui. Nam adipiscing ligula at lorem. Vestibulum gravida ipsum iaculis
justo. Integer a ipsum ac est cursus gravida. Etiam eu turpis. Nam laoreet
ligula mollis diam. In aliquam semper nisi. Nunc tristique tellus eu
erat. Ut purus. Nulla venenatis pede ac erat.
```

```
...
```

上例用于创建具有语法名称的新文件以创建新语法，本例为latin。对应的Vim文件为latin.vim，各位可以创建在自己的Vim运行时目录$HOME/.vim下。然后，使用syntax keyword命令创建一些关键字以开始语法定义。选择lorem、dolor、nulla、lectus作为关键字，语法文件可从下面这行开始：

```
syntax keyword identifier lorem dolor nulla lectus
```

编辑loremipsum.latin时，不会出现任何语法高亮显示。在自动高亮显示前，还有许多工作需要完成。但眼下我们先激活语法文件：

```
:set syntax=latin
```

因为$HOME/.vim目录是runtimepath选项里的目录之一，文本将类似图14-25所示。

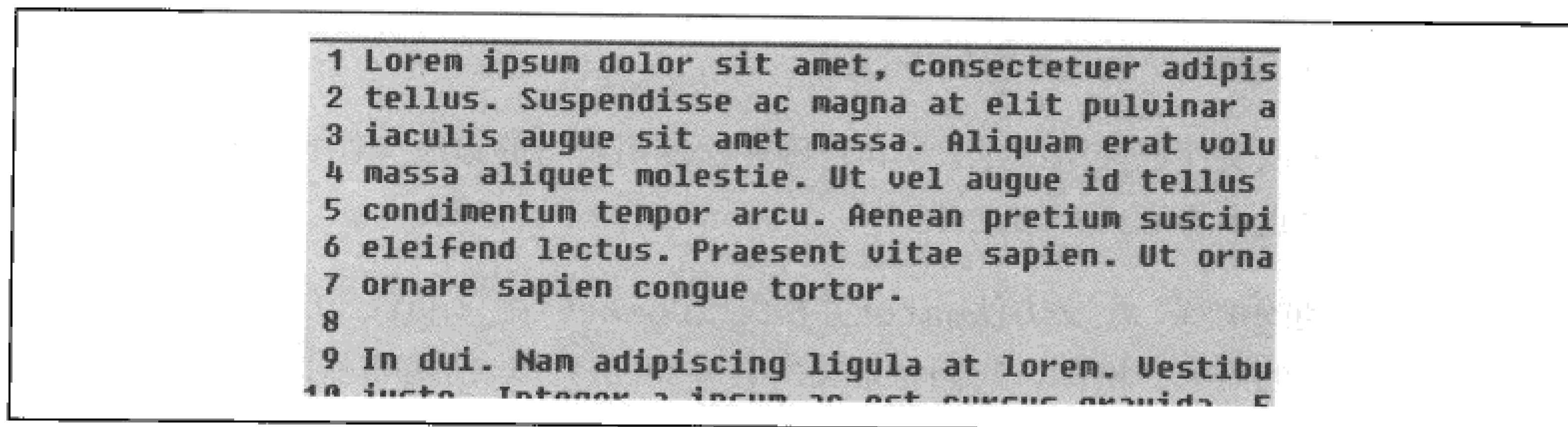


图14-25：带有关键字定义的latin文件

这有点不好辨认，我们刚定义的关键字在图中都变成了深灰色，而不再是黑色，这与其他文本的颜色不同（屏幕上的实际颜色是黑色文本与蓝色关键字）。

各位或许发现第一个出现的Lorem并未高亮显示。默认情况下，syntax keyword有大小写之分。请回到语法文件，在最顶端加上：

```
:syntax case ignore
```

这时就应该看到Lorem被包含到高亮显示的关键字中了。

再次尝试前，先把一切都改为自动化。在Vim尝试检测完所有文件类型后，它会选择性地检查其他定义，甚至可能覆盖一些定义（不推荐这么做），检查的位置在runtimepath的ftdetect目录里。因此，在$HOME/.vim下创建前述目录并创建latin.vim文件，其中包含一行：

```
au BufRead,BufNewFile *.latin set filetype=latin
```

这一行告诉Vim，任何后缀是.latin的文件都是latin文件，因此Vim应该执行在 $HOME/vim/syntax/latin.vim中的语法文件，用于显示latin文件。

现在，回头编辑loremipsum.latin，结果将如图14-26所示。

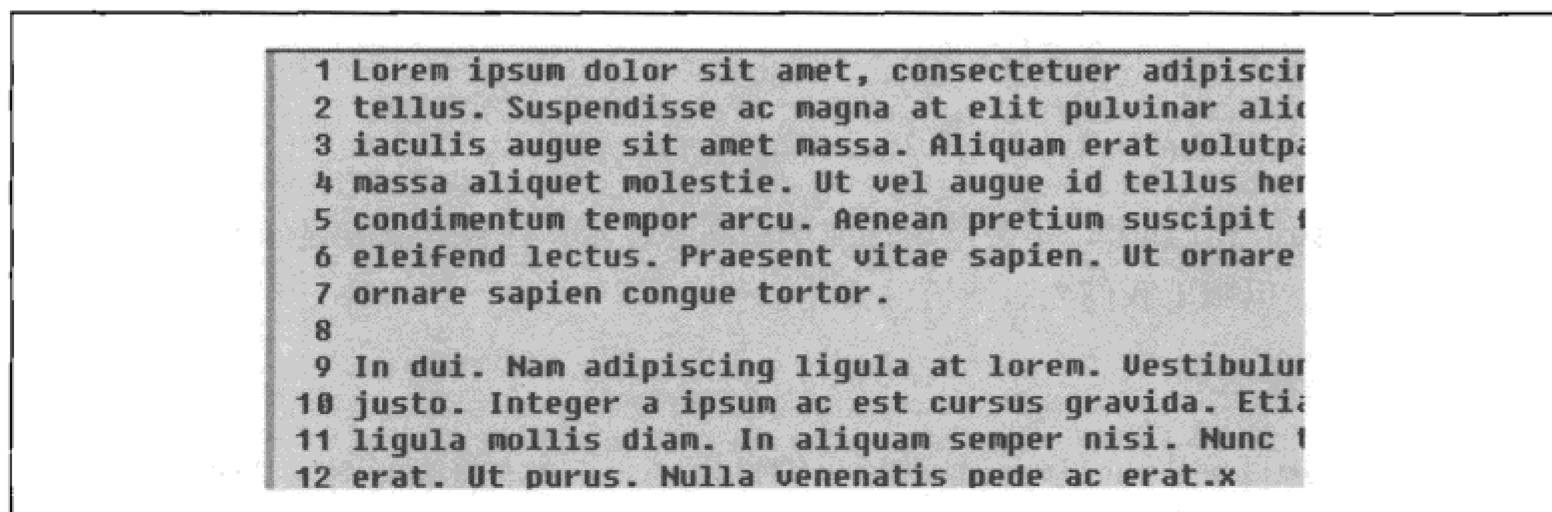


图14-26：忽略大小写的带有关键字的latin文件

首先，请注意语法文件已经激活了，Vim正确地检测出新增的语法文件类型latin。而且在匹配关键字时也不再有大小写之分。

为了做一些有趣的扩展，定义方法match并指派给Comment组。match方法使用正则表达式定义高亮显示的对象。例如，我们将定义所有以s开头并以t结尾的单词为Comment语法（这只是个范例！）。我们的正则表达式是\<s[^\t]*t\>（相信我们）。同时将定义范围为Number，范围使用正则表达式start与end定义。

我们的范围从Suspendisse开始，到sapien\.结束。若要多加一点变化，则我们决定使关键字lectus也包括在范围里。现在，latin.vim修改为如下所示：

```
syntax case ignore
syntax keyword identifier lorem dolor nulla lectus
syntax keyword identifier lectus contained
syntax match comment /\<s[^\t ]*t\>/
syntax region number start=/Suspendisse/ end=/sapien\./ contains=identifier
```

现在，编辑loremipsum.latin时，将看到图14-27所示画面。

有几件事要注意，如果各位动手制作了范例并观察呈现的配色结果，相信会比较容易看出变化：

- 出现新的高亮显示。第一行，sit高亮显示为蓝色，因为它满足match所用的正则表达式。
- 新的范围高亮显示也出现了。段落中以Suspendisse开头，直到sapien.的部分都高亮显示为紫红色（恶心）。

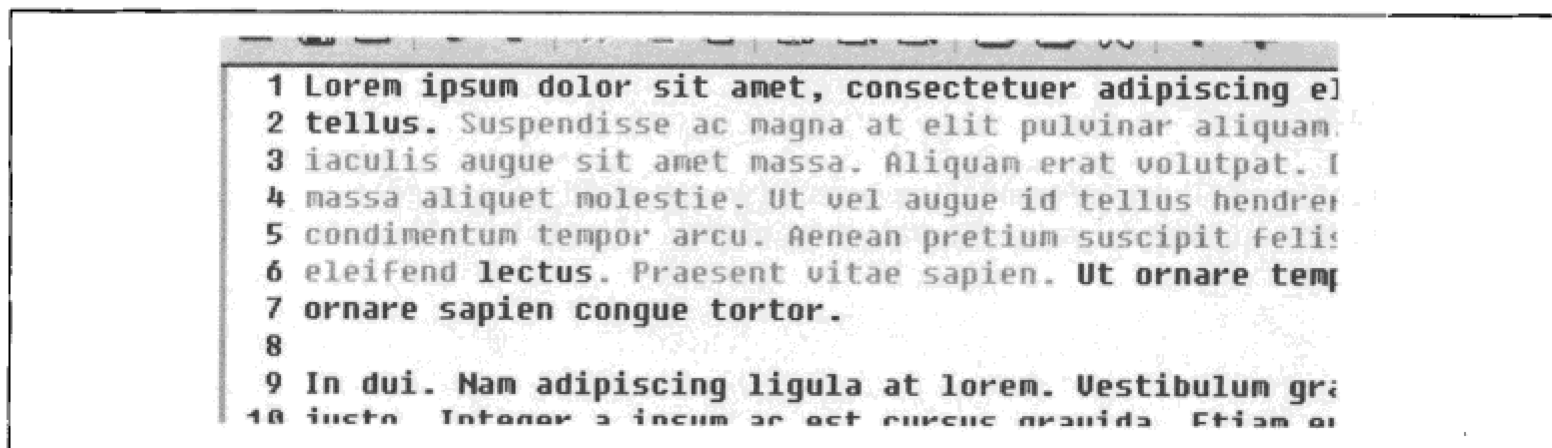


图14-27：新的latin语法高亮显示

- 关键字的语法高亮显示仍与稍早相同。
- 在高亮显示的范围内，关键字lectus仍然高亮显示为绿色，因为我们把identifier组定义为contained，并定义我们的范围为contains identifier。

这个范例才刚刚沾到语法高亮显示威力的一点点边而已。虽然这个特殊范例没什么用，但我们希望它有足够的说服力，并能激励大家尝试自行创建语法定义。

用Vim编译与检查错误

Vim不是集成开发环境（Intergated Development Environment,IDE），但它试着减轻程序员的负担，把编译功能加入编辑会话，并提供寻找与更正错误的快速简易方式。

除此之外，Vim提供了一些便利功能，用于文件中的位置跟踪与导航。我们来讨论一个简单的范例：使用Vim的内置功能及一些相关命令与功能，还有便利功能来处理“编辑—编译—编辑”周期。这些行为都要使用同一个Vim Quickfix List窗口。

作为简单的着手点，Vim让我们每次改变文件时都可使用make编译文件。Vim使用默认行为管理构建文件后的结果，这样我们才能轻易在编辑与编译阶段间切换。编译错误出现在特殊的 Vim Quickfix List窗口，可在其中查看错误、移动至错误处并更正错误。

关于这个主题，我们使用一个产生Fibonacci序列的C程序。正确且可编译的代码版本如下所示：

```
# include <stdio.h>

int main(int argc, char *argv[])
  {
  /*
   * arg 1: starting value
   * arg 2: second value
   * arg 3: number of entries to print
   *
```

```
*/

if (argc   - 1 != 3)
  {
  printf   ("Three command line args: (you used %d)\n", argc);
  printf   ("usage: value 1, value 2, number of entries\n");
  return   (1);
  }



/* count = how many to print */
int count = atoi(argv[3]);

/* index = which to print */
long int index;

/* first and second passed in on command line */
long int first, second;

/* these get calculated */
long int current, nMinusOne, nMinusTwo;

first = atoi(argv[1]);
second = atoi(argv[2]);
printf("%d fibonacci numbers with starting values: %d, %d\n", count, first,
    second);
printf("=======================================\n");

/* print the first 2 from the starter values */
printf("%d %04d\n", 1, first);
printf("%d %04d ratio (golden?) %.3f\n", 2, second, (double) second/first);

nMinusTwo = first;
nMinusOne = second;

for (index=1; index<=count; index++)
  {
  current = nMinusTwo + nMinusOne;
  printf("%d %04d ratio (golden?) %.3f\n",
            index,
            current,
            (double) current/nMinusOne);
    nMinusTwo = nMinusOne;
    nMinusOne = current;
    }
}
```

在Vim中使用如下命令编译这个程序（假设文件名为fibonacci.c）：

```
:make fibonacci
```

默认情况下，Vim把make命令传至外部shell并在特殊窗口Quickfix List中获得结果。在编译上例代码后，Quickfix List窗口画面应如图14-28所示。

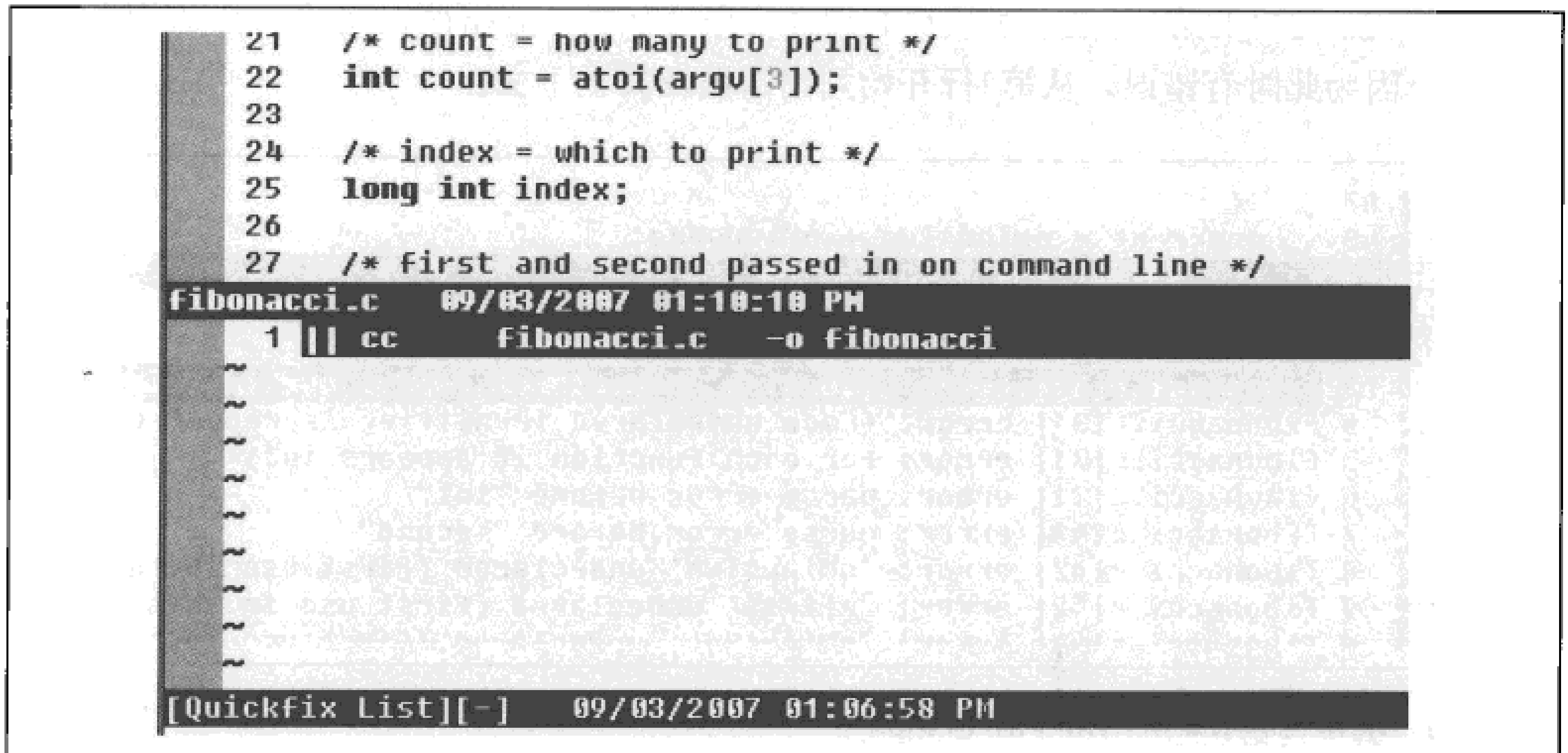


图14-28：简洁编译后的Quickfix List窗口

接下来，我们修改程序中的数行以引入一定数量的错误。

改变：

```
long int current, nMinusOne, nMinusTwo;
```

为不合格的声明：

```
longish int current, nMinusOne, nMinusTwo;
```

改变变量：

```
nMinusTwo = first;
nMinusOne = second;
```

为拼错的xfirst与xsecond：

```
nMinusTwo = xfirst;
nMinusOne = xsecond;
```

改变语句：

```
printf("%d %04d ratio (golden?) %.3f\n", 2, second, (float) second/first);
```

让它缺少一个逗号：

```
printf("%d %04d ratio (golden?) %.3f\n", 2 second (float) second/first);
```

现在重新编译这个程序。图14-29显示Quickfix List窗口现在包含的消息。

Quickfix List窗口的第1行显示了执行的编译命令。如果没有错误，这就是窗口里唯一的一行。但因为此时有错误，从第3行开始列出错误及其上下文。

```
 47      {
 48      current = nMinusTwo + nMinusOne;
fibonacci.c   09/03/2007 01:31:48 PM
  1 || cc      fibonacci.c    -o fibonacci
  2 || fibonacci.c: In function `main':
  3 fibonacci.c|31| error: `longish' undeclared (first use in this
  4 fibonacci.c|31| error: (Each undeclared identifier is reported
  5 fibonacci.c|31| error: for each function it appears in.)
  6 fibonacci.c|31| error: parse error before "int"
  7 fibonacci.c|40| error: parse error before "second"
  8 fibonacci.c|42| error: `nMinusTwo' undeclared (first use in th
  9 fibonacci.c|42| error: `xfirst' undeclared (first use in this
```

图14-29：编译有错误后的Quickfix List窗口

Vim在Quickfix List窗口中列出所有错误并让用户取用源代码，源代码中以数种方式提示错误。Vim在Quickfix List窗口中按惯例高亮显示第一个错误。然后在源文件中移动光标位置（需要时即滚动屏幕），把光标放在源代码中与错误相应的行的开始处。

在修正错误时，有数种方式能移动光标至下一个错误：输入命令:cnext；或在Quickfix List窗口中把光标移到有错误的行，然后按下ENTER。同样地，Vim会在需要时滚动源文件，并把光标放在有问题的源代码行的开始处。

在做了修正并对修正错误后的结果满意后，使用相同技巧重新开始“编译—编辑”的周期。如果你有标准开发环境（Unix/Linux机器几乎都已经如此），Vim的默认行为就是以前述方式处理“编辑—编译—编辑”，不会有什么突发状况。

如果Vim的默认行为并未发现适当的编译程序，它还有可以用于定义实用程序位置的选项，以便让我们做该做的事。关于编程环境与编译器的细节，已经超出本书的讨论范围，但我们仍列出这些Vim选项，如果各位需要研究开发环境时，可将此作为着手点：

makeprg
: 包含开发环境的make或compile程序的名称的选项。

:cnext, :cprevious
: 分别移动光标至下一个或前一个错误位置的命令，如Quickfix List窗口中的定义。

:colder, :cnewer
: Vim 能保存最后10个错误列表。这两个命令分别于Quickfix List窗口中载入较旧的前10个错误及较新的后10个错误列表。每个命令均可以接受可选的整数*n*以载入第*n*旧或第*n*新的错误列表。

errorformat

此选项定义Vim匹配所用的格式，以找出编译程序返回的错误。Vim的内置说明文档中，关于此选项的定义有更详细的信息，但默认值已可运作良好。如果你需要调整这个选项，请参阅它的细节：

```
:help errorformat
```

Quickfix List窗口的更多运用

Vim也允许我们在文件里创建自己的位置列表，通过类似grep的语法指定位置。Quickfix List窗口返回我们要求的结果，返回格式则紧密组合了（稍早所提）编译进程返回的诸行。

对本次的范例，我们在 DocBook（一种类似 XML 的形式）中编写脚本。在编写过程中的某个时间点，我们把任何包含“vim”的表示法，都从<emphasis>转换为<literal>。所以，如下事项：

```
<emphasis>vim</emphasis>
```

会被改变为：

```
<literal>vim</literal>
```

在执行下列命令后：

```
:vimgrep /<emphasis>vim<\/emphasis>/ *.xml
```

Quickfix List窗口包含了图14-30显示的信息。

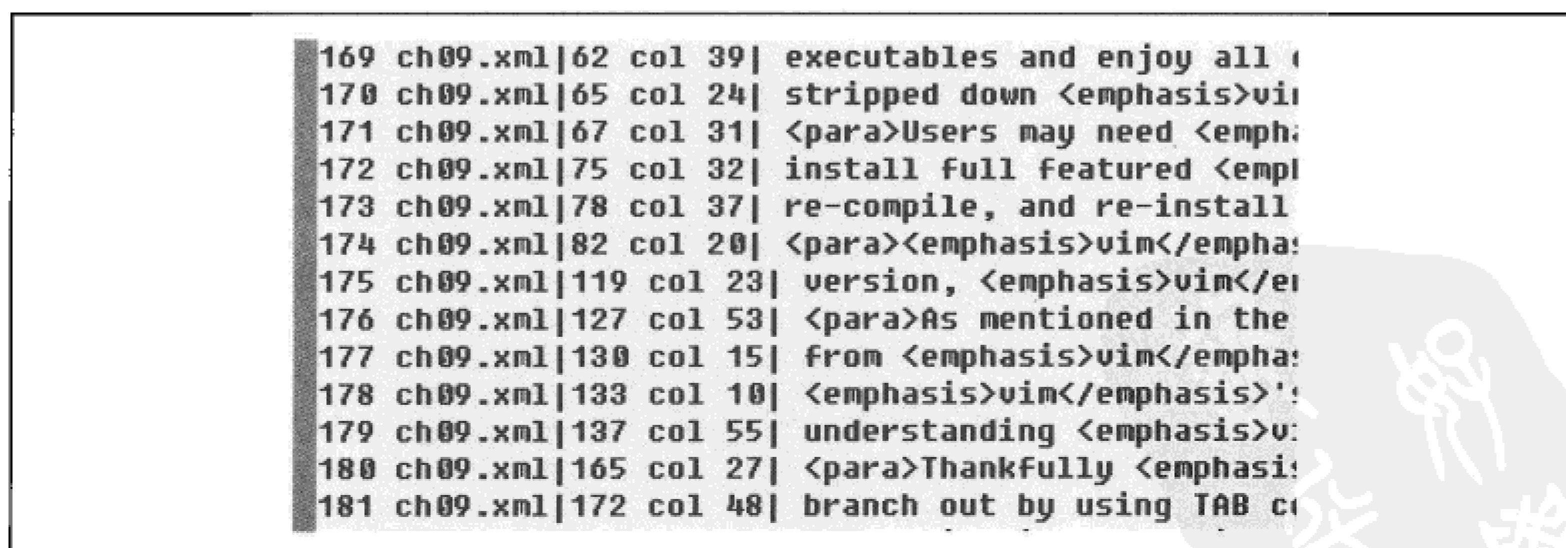


图14-30：执行：vimgrep命令后的Quickfix List窗口

接下来，要在所符合的事项间移动以及改变成新的值就很简单了。

注意：这个范例看来或许更简单地解决了一个问题，稍早我们用下列命令处理过：

```
:%s/<emphasis>vim<\/emphasis>/<literal>vim<\/literal>/g
```

但请记住，vimgrep的使用更为普遍，而且是对多个文件操作。本例是对vimgrep能力的示范，而不是实行此项任务的唯一方式。在Vim中，同一项任务通常都有好几种完成方式。

关于使用Vim设计程序的最后叮咛

本章我们已看到不少Vim的威力强大的功能。花一点时间熟练相关技巧，将为你带来极大的生产力。如果你一直都在使用vi，表示你已经爬过陡峭的学习曲线。学习Vim的额外功能，值得各位再付出一次努力，再挑战一次学习曲线。

如果你是位程序员，我们希望这章表达出了Vim能为你的任务带来多少帮助。希望大家尝试一些功能，甚至动手依据自己的需求来扩展Vim。也许你能打造出足以反馈给Vim社群的扩展组件。好了，开始设计吧！

第十五章

其他好用的Vim功能

第十章到第十四章涵盖了强大的Vim功能与诀窍，我们认为这些功能与诀窍对于有效率地使用这个编辑器是不可或缺的。本章则用比较轻松的眼光看Vim。本章综合了不属于前几章的一些主题、关于编辑的想法、Vim的哲学，还有一些好玩的Vim信息（这可不是说前几章就不好玩哦）！

编辑二进制文件

正规而言，Vim就像vi，是个文本编辑器。但Vim也能让我们编辑包含一般人看不懂的数据的文件。

为什么我们想编辑二进制文件？二进制文件之所以是二进制，不是有它存在的道理吗？二进制文件通常不都是由应用程序产生，且定义良好又有特殊格式吗？

警告：我们固然享受Vim的二进制编辑功能，但此处并不深入讨论编辑二进制文件时需要考虑的潜在议题。例如，有些二进制文件包含数字签章checksum，以确保文件的完整性。编辑这些文件要冒着损害其完整性的风险，而且可能让文件不能使用。因此，别把本节当成编辑二进制文件时的靠山。

没错，二进制文件通常由计算程序或模拟过程创建，而且并非用于手动编辑。例如，数码相机通常把相片存储为JPEG格式，一种数码相片采用的压缩二进制格式。这种文件是二进制文件，但具有定义良好的段落或块，存储着标准信息（也就是说，如果根据规范实现，即可存储信息）。JPEG格式的数码相片把相片的元信息（meta information，例如拍照的时间、分辨率、相机设置、日期等等）存储在保留块里，与压缩的数码相片数据分隔。实际的应用程序或许会使用Vim的二进制文件编辑功能，以编辑某个目录下的JPEG文件，更改所有“创建”块里的年份字段，来更正相片的“创建日期”字段。

图15-1显示了JPEG文件的编辑会话。请注意光标位于日期字段上。改变这些字段，我们即可直接编辑关于这张图片的信息。

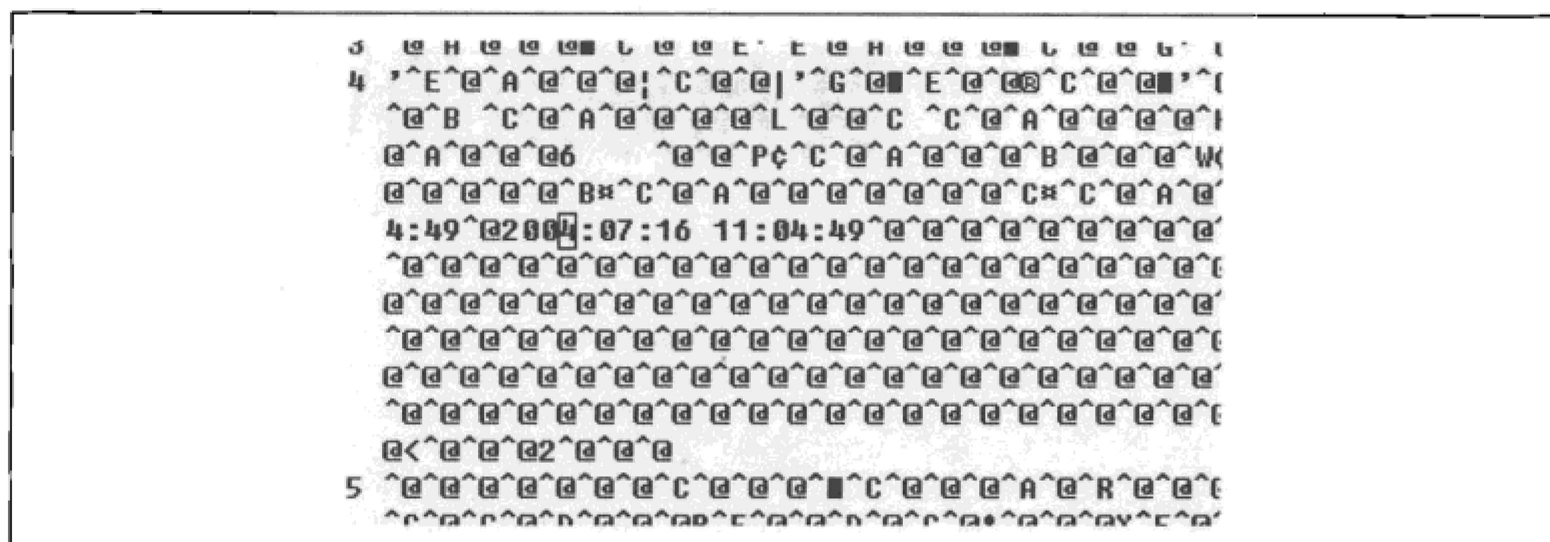


图15-1：编辑一个二进制JPEG文件

对于熟悉特定二进制格式的专家级用户，Vim可以非常方便地直接改变二进制文件，否则可能需要冗长、重复地使用其他工具访问文件。

主要有两种编辑二进制文件的方式。在Vim命令行中可以设置binary选项：

```
set binary
```

或在启动Vim时加上-b选项。

为了辅助二进制编辑并避免破坏文件的完整性，Vim设置下列选项：

- textwidth与wrapmargin设置为0。以免Vim在文件中插入伪造的newline序列。
- modeline与expandtab均不予设置（nomodeline与noexpandtab）。以免Vim用tab增加shiftwidth的空间，并避免Vim在模式行（modeline）里解释命令，否则有可能设置某些选项而造成意外副作用。

注意： 使用二进制模式时，在窗口间移动或在缓冲区间移动时务必小心。Vim使用entry与exit事件设置并改变切换缓冲区与窗口的选项，或许有人会误以为这是删除某些刚列出的保护措施。我们推荐在编辑二进制文件时使用单一窗口、单一缓冲区。

digraph：非ASCII字符

什么？《Messiah》的作者是George Frideric *Hädel*，而不是 George Frideric *Handel*？“*réumév*好像比*resume*多了一些特征？请使用Vim的digraph（二合字母）输入特殊字符。

即使是英文文本，有时也需要特殊字符。尤其是在这个全球化的世界里引用其他语言的文献时。非英文的文本文件需要大量特殊字符。

Vim 有好几种输入特殊字符的方式，其中两种相对简单与直观。这两种方式分别依赖通过前缀（CTRL-K）及在两个键盘字符间使用BS（Backspace）键而定义digraph（其他方法更适合以字符的原始数值插入字符，可用十进制、十六进制、八进制数指定字符。这些方法虽然威力强大，却不容易记忆数值及其对应字符）。

注意：*digraph*一词，传统上用于描述结合两个字母的字符，代表单一音节，例如“digraph”或“phonetic”中的*ph*。Vim借用结合两个字母的表示法，以描述特殊字符的输入机制，这些字符包括具有特殊特征、典型的重音记号，或像“ä”所用的其他元音变音记号。这些特殊记号的正确称呼是*diacritic*或*diacritical mark*（区别发音符号）。换句话说，Vim使用digraph创建diacritic。

第一种输入diacritic的方法，是由三个字符组成的序列：CTRL-K、基本字母与一个标点符号字符，用于指示重音或需要增加的记号。例如，想创建附有变音符号的c（ç），输入CTRL-Kc,；想创建附有重音记号的 a（à），输入CTRL-K a!。

希腊字母能用相对应的拉丁字母后接星号来创建（例如输入CTRL-Kp*可输出小写的π）。俄文字母能用相对应的拉丁字母后接等号创建（有少数情况需使用百分比符号）。使用CTRL-K?I（确定使用了大写的I），再输入反向的问号（¿）与CTRL-K ss，则可输入德文字母sharp S（ß）。

第二种输入特殊字符的方式，则需设置digraph选项：

```
set digraph
```

现在，输入双字符组合中的第一个字符，然后加上退格键（BS），再输入创建记号的标点符号。因此，ç的输入方式是 cBS，à的输入方式则是aBS!。

设置digraph选项，不会阻碍各位使用CTRL-K。如果不常输入特殊字符，可考虑只使用CTRL-K的方式。否则，有可能在按下退格键更正输入错误时，发现自己经常不小心输入了digraph。

使用:digraph命令显示所有默认字符序列；若需详尽说明，则可使用:help digraph-table 取得。图15-2呈现digraph命令的部分结果。

图中每个digraph都用三个字段表示。看起来很乱，因为Vim在画面的一行上能塞入多少个三栏组合，就塞入多少组合。在每组中，第一栏显示了digraph的双字符组合，第二栏显示组合代表的digraph，第三栏则是digraph的十进制Unicode值。

```
SH ^A    1  SX ^B    2  EX ^C    3  ET ^D    4
UT ^K   11  FF ^L   12  CR ^M   13  SO ^N   14
NK ^U   21  SY ^V   22  EB ^W   23  CN ^X   24
US ^_   31  SP      32  Nb #    35  DO $    36
(! {   123  !! |   124  !) }   125  '? ~   126
NL ■   133  SA ■   134  ES ■   135  HS ■   136
S3 ■   143  DC ■   144  P1 ‘   145  P2 ’   146
GC ■   153  SC ■   154  CI ■   155  ST ■   156
Pd £   163  Cu ¤   164  Ye ¥   165  BB ¦   166
-- -   173  Rg ®   174  'm ¯   175  DG °   176
.M ·   183  ',  ¸  184  1S ¹   185  -o º   186
A' Á   193  A> Â   194  A? Ã   195  A: Ä   196
E: Ë   203  I! Ì   204  I' Í   205  I> Î   206
O? Õ   213  O: Ö   214  *X ×   215  O/ Ø   216
ss ß   223  a! à   224  a' á   225  a> â   226
e' é   233  e> ê   234  e: ë   235  i! ì   236
o' ó   243  o> ô   244  o? õ   245  o: ö   246
```

图15-2：Vim digraphs

为了大家使用方便，表15-1列出最常用到的重音与记号及代表它们的组合中最后出现的标点符号。

表15-1：如何输入重音与其他记号

记号（范例）	表达digraph的字符
尖重音符（fiancé）	单引号（'）
短音符号（publică）	左括号（()
楔形符点（Dubček）	小于号（<）
变音符号（français）	逗号（,）
抑扬符号（português）	大于号（>）
沉音符号（voilà）	感叹号（!）
长音符号（ā tm ā）	连字符（-）
斜删除线（Søren）	斜线（/）
波浪符号（señor）	问号（?）
曲音符号（Noël）	冒号（:）

在其他地方编辑文件

感谢网络协议的无缝集成，Vim接受我们编辑远程机器上的文件，就像文件在本地一样！如果你只简单指定URL作为文件，Vim将在窗口中打开该文件，并把改变写入远程系统里（但你必须有写入的权限）。举例来说，下列命令编辑一份系统mozart上的用户

ehannah拥有的.vimrc文件。远程机器于port 122提供SSH安全协议（port 122是个非标准端口，通过隐匿而增加安全性）：

```
$ vim scp://ehannah@mozart:122//home/ehannah/.vimrc
```

因为我们在远程机器上编缉ehannah的主目录下的文件，可使用简单文件名而缩短URL，故它被视为相对于远程系统上用户主目录的路径：

```
$ vim scp://ehannah@mozart:122/.vimrc
```

让我们分解URL，以便学习如何针对特定环境建立URL：

scp:
: 第一个部分，到冒号为止，代表传输协议。本例的协议为scp，建立在Secure Shel（SSH）协议上的文件复制协议。一定要加冒号（:）。

//
: 这个部分引入主机信息，对大部分传输协议均采用[*user*@]*hostname* [:*port*]的形式。

ehannah@
: 可选部分。对scp这类安全协议，这个部分指定登录远程机器时的用户身份。省略时，默认值为你在本地机器上的用户名称。收到要求提供口令时，你必须输入该用户在远程机器上的口令。

mozart
: 这部分是远程机器的符号名称，也能用数值地址表示，例如192.168.1.106。

:122
: 可选部分，指定协议使用的端口。冒号分隔端口编号与前面的主机名称。所有标准协议都使用熟悉的端口，若使用标准端口，可在URL中省略这个部分。本例中，122并非scp协议的标准端口，而且因为mozart系统的管理器选择通过port 122提供服务，所以必须予以指定。

//home/ehannah/.vimrc
: 这是远程机器上我们想编辑的文件。开始处有两个斜线，因为指定的是绝对路径。相对路径或简单文件名只需要一个斜线，用以和前面的主机名称区别。相对路径与（我们登录的）用户的主目录相对。所以，本例中的相对路径将与ehannah的主目录相对，例如/home/ehannah。

以下列出几种得到支持的协议：

- *ftp:*与sftp（一般FTP与安全FTP）

- *scp:*（通过 SSH 的安全远程复制）
- *http:*（使用标准浏览器协议传送文件）
- *dav:*（相对新，但深受欢迎的网络传递公开标准）
- *rcp:*（远程复制）

到当前为止说明的内容已足以实现远程编辑，但这个过程可能不像在本地进行文件编辑一样透明。也就是说，因为其间对于移动远程主机数据的需求，各位可能需要口令才能做这项工作。如果习惯在编辑期间定时把文件写入磁盘，这种对口令的要求可能会很烦人，每次“写入”都会跳出输入口令的提示窗口，正确输入才能执行动作。

前述所有传输协议均可调整服务的配置，以允许不需口令的访问，但详细情况各异。请参考各项服务的说明文档，以了解特定协议的相关细节及配置。

目录间的移动与改变

如果经常使用Vim，或许你已经意外发现可以查看目录并在目录间移动，只需使用与在文件内移动相似的按键方式。

假设有一个包含许多`.c`文件的目录`ex-050325`（刚好也是包含`vi`的可编辑源代码的目录）。下列命令可编辑ex-050325：

```
$ vim ex-050325
```

图15-3截取了命令执行后的部分画面，各位看到的结果应该与此相似。

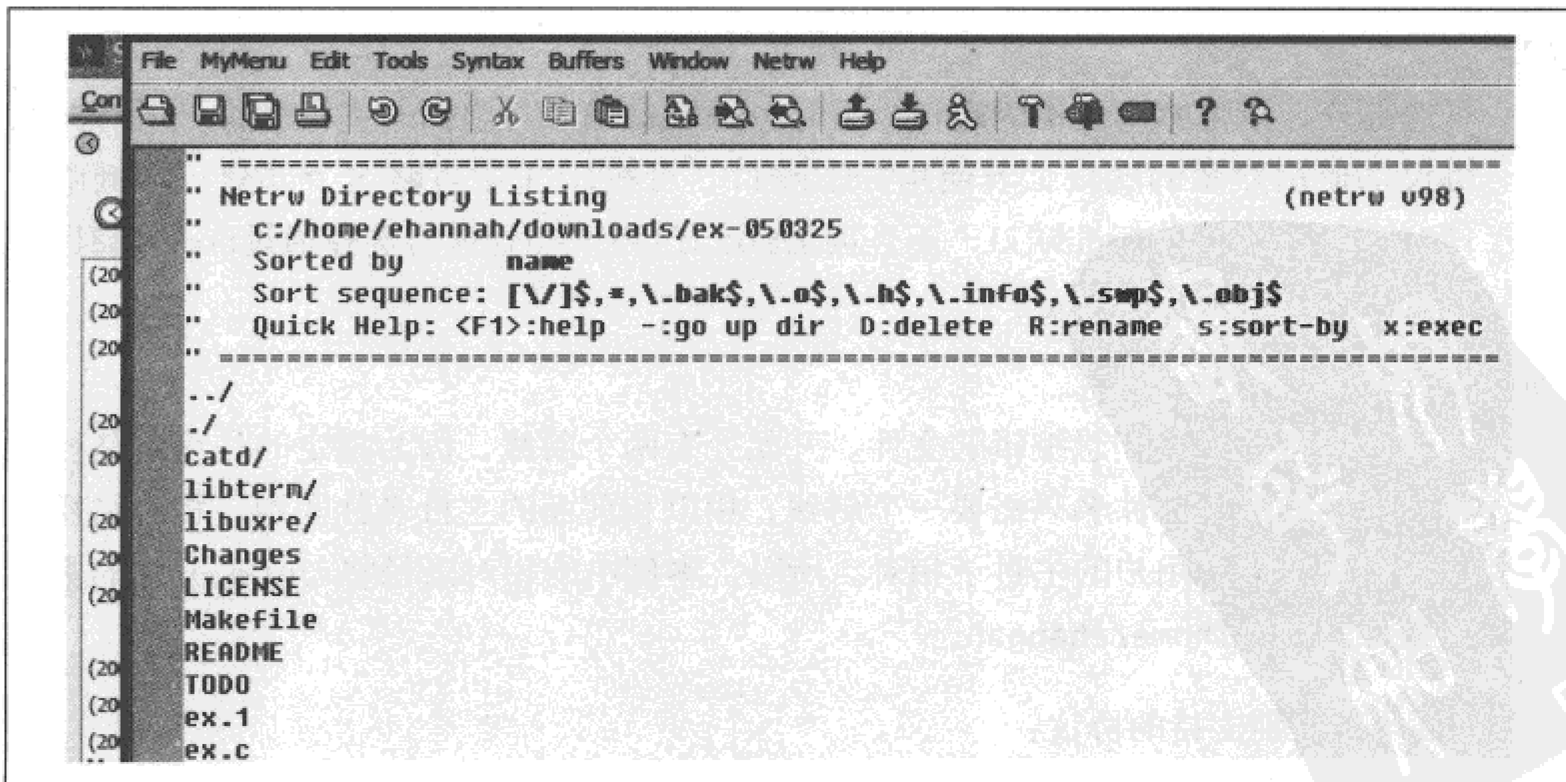


图15-3：Vim“编辑”ex-050325目录

Vim显示三种信息：概述性质的注释（位于等号构成的横线前）、目录（以斜线表示）、文件。每个目录或文件均自成一行。

有许多使用这项功能的方式，但使用标准Vim移动命令（例如w是移到下个词、j是跳到下一行）或在所需条目上点击鼠标，是立即且直接带出生产力的最省力方式。目录模式的特殊功能包括：

- 当光标移到目录名称上时，按下ENTER键即可转到该目录。
- 当光标移到文件名称上时，按下ENTER键即可编辑该文件。

注意： 如果想保持目录窗口的打开，以便进一步操作目录，则在编辑（光标所在位置）文件时按下o，Vim将分割窗口，让我们于新创建的窗口中编辑文件（这在移动至另一个目录时也很好用，Vim同样分割窗口，并于新窗口中"编辑"刚才移入的新目录）。

- 可以删除或重新命名文件与目录。键入大写的R即可重新命名文件或目录。或许有点不直观，Vim可创建一个命令行提示符，以执行重新命名，如图15-4所示。

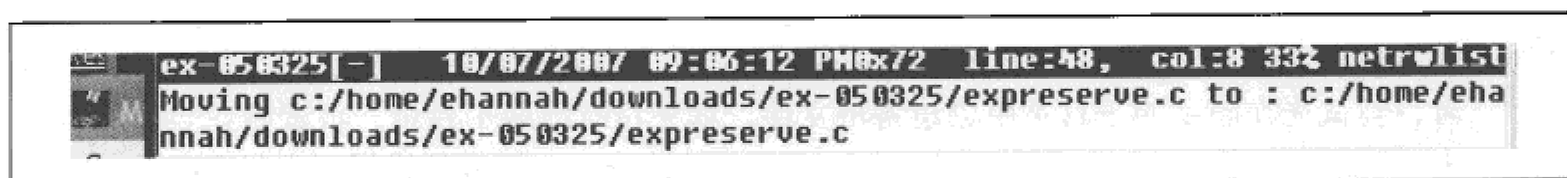


图15-4：用于"编辑目录"中的重命名的提示符

欲完成重新命名，请编辑第二个命令行参数。

删除文件的方式也很相近。只要把光标放到欲删除的文件名下，然后键入大写的D。Vim会弹出确认要删除文件的对话框。与重新命名的功能一样，Vim的确认提示符位于屏幕上的命令行。

- 有项对目录编辑的很好改进——通过Vim的搜索功能而快速访问文件。假设想编辑ex-050325目录下的expreserve.c文件。若欲快速找到并编辑它，可搜索部分文件名：

```
/expreserve.c
```

并在光标移到文件名时，按下ENTER或 o。

注意： 阅读关于目录编辑的联机帮助时，各位将看到Vim把这个功能归在使用网络协议编辑文件的内容里，也就是前一节提到的内容。我们则把目录编辑独立列成一节。因为它很有用，与庞大的网络协议编辑文件内容放在一起时很容易忽略它。

使用Vim备份

Vim协助我们不要意外地损失文件，它让我们制作编辑中文件的备份。对于会发生可怕问题的编辑会话而言，这项功能非常地有用，因为我们可以恢复稍早的文件。

备份行为由两个选项控制：backup与writebackup。创建备份的位置与方式则由另外4个选项控制：backupskip、backupcopy、backupdir、backupext。

如果backup与writebackup选项都设为关闭（也就是nobackup与nowritebackup），则Vim不会为编辑会话做备份。如果打开backup，Vim会删除任何原有备份，并为当前的文件创建备份。如果backup关闭但writebackup打开了，则Vim为编辑会话创建备份文件，删除备份文件则留到以后再做。

backupdir是以逗号分隔的目录列表，列出了Vim创建的备份文件的位置。例如，如果想在系统的临时目录中创建备份文件，可以把backupdir设为"C:\TEMP"（Windows适用）或"/tmp"（Unix与Linux适用）。

注意： 如果想让备份文件总是存储在当前编辑的目录中，可以指定备份目录为“.”（点号）。或者，尝试先在隐藏的子目录中创建备份文件，如果没有隐藏子目录，再试着在当前的目录中创建：方法是定义backupdir值为类似"./.mybackups,."这样的设置（点号表示文件当前所在的目录）。这是个很有灵活性的选项，支持许多定义备份位置的策略。

如果想在编辑会话创建备份但不想为所有文件创建，则可使用backupskip选项定义模式列表（以逗号分隔）。Vim不会为匹配模式的任何文件创建备份。例如，不想为任何在/tmp或/var/tmp目录下编辑的文件做备份时，设置backupskip为"/tmp/*,/var/tmp/*"即可阻止Vim为我们备份该类文件。

默认情况下，Vim创建的备份文件名与原始文件一样，但备份文件的后缀是 ~。这是个比较安全的后缀，因为文件名很少以这个字符结尾。使用backupext选项，可改变你想要的后缀。例如想让备份文件的后缀是.bu，请设置backupext为字符串".bu"。

最后一个选项backupcopy定义了创建备份文件的方式。我们推荐设置选项为"auto"，让Vim计算备份的最佳方式。

以HTML表现文本

曾经需要对一群人展现你的代码或文本内容吗？曾经试着使用其他人的Vim配置查看代码，结果不能搞清楚内容吗？可以考虑把你的文本或代码转换成 HTML并通过浏览器查看。

Vim提供三种创建文本的HTML版本的方式。三种方式都使用与原始文件相同的名称，加上后缀.html创建新缓冲区。Vim会分割当前会话窗口并显示HTML版本的文件于新窗口：

gvim的*“Convert to HTML”*

这是最友好的方式，建立在gvim图形编辑器里（请参见第十三章）。打开gvim中的Syntax菜单并选择“Convert to HTML”。

2html.vim*脚本*

这是前面的“Convert to HTML”项调用的底层脚本。可直接以下列命令调用：

```
:runtime!syntax/2html.vim
```

它不接受范围的指定，而会转换整个缓冲区。

TOhtml*命令*

这个方式比2html.vim脚本有灵活性，因为我们可以指定一段范围，只指定想转换的行。例如要转换缓冲区中的第25行到第44行，请输入：

```
:25,44TOhtml
```

将gvim转换成HTML的好处在于，图形用户界面（GUI）能准确地检测色彩并创建正确对应的HTML指令。这些方式也能在非GUI的上下文中运作，但较不能确保其执行结果的准确，且或许派不上用场。

注意： 管理新创建文件的方式要看各位的选择。Vim不为我们保存文件，只是创建缓冲区而已。我们推荐使用某种管理政策，以保存HTML文件并同步文件的HTML版本。例如，创建一些自动命令来触发HTML文件的创建与保存。

已保存的HTML文件能使用任何网络浏览器查看。或许有人不太熟悉在本地系统上使用浏览器打开文件的方式，其实这非常简单，几乎所有浏览器都在它的File菜单中提供了Open File选项，可呈现文件选择对话框，让你选择包含HTML文件的文件夹。如果你计划经常使用这项功能，我们推荐为你的所有文件创建一份书签。

有何差异？

同一个文件的两个版本，其间差异通常很少，如果有个只让我们看一眼就能知道两个版本间差异的工具，想必能省下工作时间。Vim集成了Unix世界中著名的diff命令，通过vimdiff命令调用非常复杂的可视界面。

有两种方式可调用这项功能：独立命令或Vim的选项：

```
$ vimdiff old_file new_file
$ vim -d old_file new_file
```

通常，比较版本时的第一个文件是较旧的版本，第二个文件则是较新的版本，但这只是惯例。事实上，就算调换顺序也可以完成比较。

图15-5示范`vimdiff`的输出结果。因为屏幕画面有限，我们压缩宽度并关闭 Vim 的`wrap`选项，以便看出差异之处。

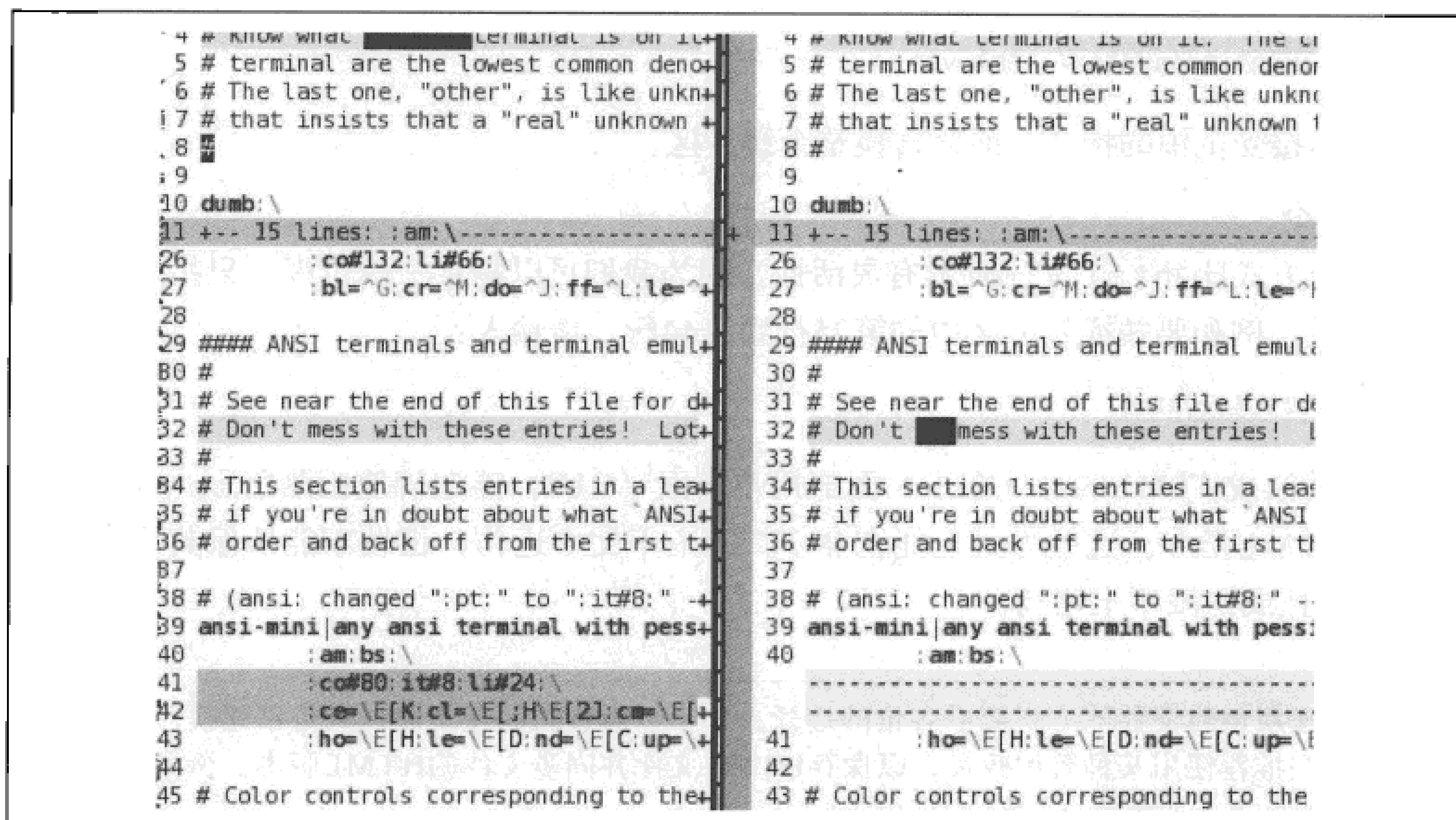


图15-5：vimdiff的结果

虽然示范图不能传达出视觉内容的完整效果（尤其是彩色只能用灰度表示），但它表达出了关键行为：

- 在第4行，可以看到左窗口里的一个黑色方块并未出现在右窗口。这是个高亮显示的词汇，指出这一行在两个版本间的不同。相似地，在第32行，右窗口里有个高亮显示的词汇，但左窗口中没有。
- 在第11行，在两个窗口里，Vim都创建了15行的折叠。因为在这两个版本中，这15行完全一样，所以Vim创建折叠，让屏幕能显示多一些有用的“diff”信息。
- 第41行到第42行，在左窗口里是高亮显示的，在右窗口里的相应位置则是虚线，表示这两行不见了。从这里开始，两个版本的行编号不再一样，因为右窗口中少了两行，但两个版本中相应的行仍然水平并排。

所有类似 Unix 的Vim版本都已附带有vimdiff，因为diff命令是Unix的标准命令。非Unix的Vim版本应该附有Vim自己的diff。Vim允许替换diff命令，只要命令能创建标准的diff输出。

diffexpr变量定义替换默认vimdiff行为的表达式，并通常实现成操作下列变量的脚本：

v:fname_in
: 进行比较的第一个输入内容

v:fname_new
: 进行比较的第二个输入内容

v:fname_out
: 捕捉diff输出结果的文件

撤销“撤销”

除了可撤销任意数量的编辑行为外，Vim还提供一个有趣的技巧，称为 *branching* undo。

要使用这项功能，首先决定对于撤销编辑操作的控制程度。使用undolevels选项定义一次编辑会话中可以撤销的改变次数。默认值为1000次，对于大多数用户，这个数量应该相当足够了。如果想与vi兼容，则设置undolevels为零：

```
:set undolevels=0
```

在vi中，撤销命令u是文件当前状态与前一次改变状态的切换开关。第一次撤销，转换到前一次改变的状态。再一次撤销，则恢复为文件撤销前的状态。而Vim的行为与此非常不一样，因此命令的实现也不一样。

Vim并非在最近一次改变间切换，若重复调用Vim的撤销功能，将从最近一次改变开始，依序撤销前面的所有改变，直到undolevels选项定义的限制。因为撤销命令u只能向后移动，我们需要一个向前移动并重做改变的命令，故Vim有一个重做改变的命令，:redo，也可用CTRL-R键。CTRL-R键前加上数值，可定义重做改变的次数。

使用撤销命令（u）与重做命令（CTRL-R）在改变间移动时，Vim会维护一份文件状态地图，并知道前一次撤销的可能执行时间。当所有可能的撤销完成后，Vim即复位文件的修改状态，并允许用户在没有添加! 后缀的状况下离开文件。虽然对最终用户交互没有太多帮助，但对于幕后的脚本设计而言，文件的修改状态就很重要了。

大多数用户只需要撤销和重做就已足够。但想想更复杂的情况：如果你对文件做了7次

改变，然后撤销了3次？到目前为止还可以，没有不寻常的事情需要考虑。但假设在撤销3次后，又做了一次改变，并与Vim的改变记录中的下一次改变不同。Vim在改变的历史记录中定义这个分歧点为分支（branch），分歧点以后，改变的路径开始不同。有了路径，我们就可以依据时间前后移动，加上分支后，即可沿着记录改变的不同路径移动。

关于在改变路径间移动的完整语句，请利用Vim的help命令获得：

```
:help usr_32.txt
```

现在的位置？

大多数文本编辑都从第1行、第1栏开始。也就是说，每次打开编辑器，文件载入及编辑都从第1行开始。但如果某个文件已被编辑很多次，有了很多内容，则编辑会话若能从上次结束编辑的地方开始，应该会更为方便。Vim能达成你的需求。

有两种方式能保存编辑会话信息备用：viminfo选项及mksession命令。

viminfo选项

Vim使用viminfo选项定义保存编辑会话信息的方式与位置。该选项是一个字符串，具有用逗号分隔的参数，告知Vim需要保存的信息量以及保存的位置。下面介绍一些viminfo的子选项：

<n

告知Vim为每个寄存器保存内容行，最大限度为*n*行。

> **注意：** 如果未指定任何值给这个选项，则每一行都会被保存。刚开始时，这种设置似乎很正常，但想想经常编辑大型文件并大量改变文件的状况。举例而言，如果经常编辑具有10 000行的文件，然后删除每一行（可能是为了外部应用程序快速添加文件内容而予以削减），再保存文件，则总共10 000行的内容将保存在viminfo文件中。如果经常这样处理多个文件，viminfo文件将变得非常庞大。然后你会发现启动Vim时出现严重延迟，因为Vim每次启动时均需处理viminfo文件。
>
> 我们推荐为这个选项进行合理且有用的限制。本章作者使用50。

/n

要保存的搜索模式记录项的数量。如果并未指定，Vim将使用history选项的值。

:n

要保存的命令行历史记录的最大数量。如果并未指定，Vim将使用history选项的值。

'*n*

Vim维护信息的最大文件数量。如果定义了viminfo选项，即需要这个参数。

以下是Vim保存在viminfo文件中的内容：

- 命令行历史记录
- 搜索字符串历史记录
- 输入行历史记录
- 寄存器
- 文件标记（例如用mx创建的标记即会被保存，并能于输入'*x*重新编辑文件时移动）
- 前次搜索及替换模式
- 缓冲区列表
- 全局变量

这个选项在维持编辑会话的持续性方面非常方便。例如编辑大型文件时，在其中改变了模式，则搜索模式与光标在文件中的位置都会被一并记忆。想在新会话中继续搜索，只需输入n，就可移动到下一个出现搜索模式的地方。

mksession命令

Vim用mksession命令保存所有关于会话（session）的编辑信息。sessionoptions选项包含以逗号分隔的字符串，指定该在编辑会话里保存的内容。这种保存编辑会话信息的方式更为全面，比viminfo更具体。这个方式可保存当前编辑会话中的所有文件、缓冲区、窗口等等。在mksession保存信息后，整个会话即可重新构建。由于所有被编辑的文件与所有选项的设置，甚至窗口尺寸都将被保存，因此重新载入信息即带来会话的重建。至于viminfo，它只以文件为单位来保存编辑信息。

想以这种方式保存会话时，输入

```
:mksession [filename]
```

其中filename指定要保存会话信息的文件。Vim创建用于重新构建会话的脚本文件与稍后执行的source命令（未指定文件名时，默认值为Session.vim）。如果以下列命令保存会话：

```
:mksession mysession.vim
```

即可使用下列命令重建会话：

```
:source mysession.vim
```

以下列出可从会话中保存的内容以及用于保存对应内容的sessionoptions选项的参数：

blank
: 空白窗口

buffers
: 隐藏与未载入的缓冲区

curdir
: 当前的目录

folds
: 手动创建的折叠、打开或关闭的折叠以及本地的折叠选项。

注意：保存手动创建的折叠以外的内容没有意义。自动创建的折叠本来就会再次自动创建！

---

globals
: 全局变量，以大写字母开头，其后至少包含一个小写字母

help
: 帮助窗口

localoptions
: 定义为本地窗口采用的选项

options
: 由:set设置的选项

resize
: Vim窗口的尺寸

sesdir
: 会话文件所在的目录

slash
: 将文件名里的反斜线替换为斜线

tabpages
: 所有分页

注意： 如果未在sessionoptions字符串中指定这个参数，则只有当前分页会被单独存储。因此我们能灵活地选择在分页层次定义会话，或于全局层次（包含所有分页）定义会话。

unix
: Unix的一行结尾的格式

winpos
: Vim窗口在屏幕上的位置

winsize
: 缓冲区窗口在屏幕上的尺寸

好了，举例来说，想保存会话以保留所有缓冲区、所有折叠、全局变量、所有选项、窗口尺寸与窗口位置等所有信息，则要将sessionoptions选项定义为：

```
:set sessionoptions=buffers,folds,globals,options,resize,winpos
```

内容行（大小）

Vim允许一行拥有几乎无限的长度。我们可以在屏幕画面上，让一行文本绕排为多行，这样观看内容时才不需要水平滚动轴；亦可只显示每一行的开头，向右滚动以查看隐藏的部分。

如果你希望屏幕上的一行只显示一行文本，请关闭wrap选项：

```
set nowrap
```

设置为nowrap后，Vim显示的字符数量就为屏幕宽度所允许的数量。请把屏幕想成观景窗：通过小小的窗口观察很宽的行。以有100个字符的行而言，其比80列宽的屏幕多了20个字符。根据显示在屏幕第1列的字符，Vim判断在这有100个字符的内容中，哪些字符不予以显示。假设屏幕第1列是该行的第5个字符，表示该行的第1到4个字符位于可视屏幕以左，因而被隐藏了。第5到第84个字符则可在屏幕上看到，后面的第85到第100个字符位于可视屏幕以右，因而也被隐藏了。

Vim在我们于很长的一行上左右移动时，管理行的显示方式。Vim左右移动时，只滚动（sidescroll）最小量的字符。可以使用如下命令设置滚动的值：

```
set sidescroll=n
```

*n*是滚动的列数。我们推荐设置sidescroll为1，因为现代个人计算机能轻易提供必需的

处理能力，让一次滚动屏幕上的一列表现流畅。如果你的屏幕慢下来，响应时间也延迟了，则或许需要调高该数值，使屏幕的重绘最小化。

`sidescroll`值定义了位移的最小量。Vim移动的距离应为足以完成任何移动命令的值。例如输入`w`可移动光标至同一行的下一个词。然而，Vim对移动的处理有点微妙。

如果下一个词只有部分可见（位于屏幕右侧），Vim将移动到词汇的第一个字符，但不会重绘整行。再次下达`w`命令，则会稍微向左移动，直到能使光标位于下个词的第一个字符并予以呈现。

我们可以用`sidescrolloff`选项控制这项行为。`sidescrolloff`定义光标向左右移动时，屏幕保持的最少列数。若定义`sidescrolloff`为10，能使光标移动到屏幕两侧边缘时，维持至少10个字符的上下文空间。现在当我们移向一行的左右两侧，Vim试着把足够的文本移入可视屏幕时，光标与屏幕两侧的距离将不会少于10列。如此调整 Vim的`nowrap`模式，或许是比较好的方式。

Vim用`listchar`选项提供便利的可视信息。设置 Vim 的`list`选项后，`listchar`定义如何显示字符。Vim对此选项提供两个设置，用以控制是否使用字符表示屏幕可视范围外是否还有内容。例如：

```
set listchars=extends:>
set listchars+=precedes:<
```

告知Vim如果屏幕可视范围以左还有更多字符时，在第一列显示<；如果屏幕可视范围以右还有更多字符时，则在最后一列显示>字符。图15-6是示意图。

```
10
11 <ext This is a very long line exceeding width of screen. text >
12
```

图15-6：nowrap模式中的一个长行

相反地，如果希望看到整行内容而不想滚动水平滚动条，则用`wrap`选项绕排内容：

```
set wrap
```

现在文件内容的显示方式如图15-7所示。

```
10
11 text text This is a very long line exceeding width of screen. t
   ext text more text than a line should ever have unless you're j
   ust doing it for the sake of an example but even in that case i
   t's an awful lot of text for just one line! :-)
12
```

图15-7：wrap模式中的一个长行

不能完整显示在屏幕上的过长文本行，在第一个位置以单一字符@呈现，直到光标与文件放置的方式能让一行完整呈现。当图15-7的光标向上移动，使第11行不能完整显示时，屏幕画面将如图15-8所示。

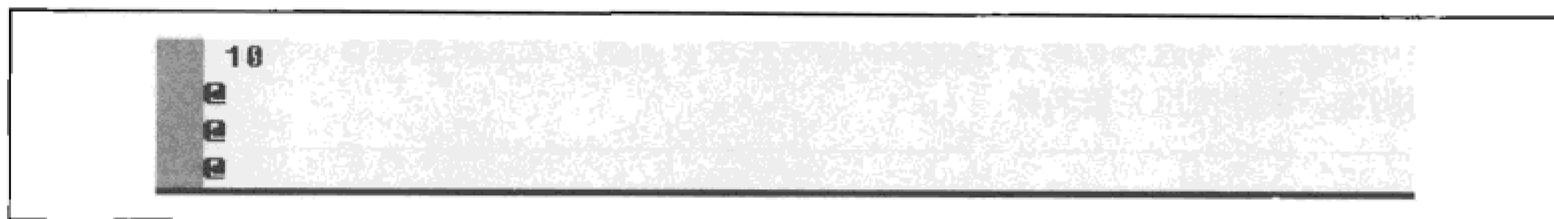


图15-8：长行指示符

Vim命令与选项的缩写

Vim中有非常多命令与选项，我们推荐先从它们的名称开始学习。几乎所有命令与选项（至少不是少数命令与选项）都有些相关的缩写，这样可以节省时间，但请确定缩写的内容！本章作者在使用缩写时，遇过某些难堪、非预料的结果——原本以为是某种结果，却变成非常不一样的东西。

随着大家越来越有经验，也会发展出自己喜爱的一套Vim命令与选项，并使用某些命令与选项的缩写以节省时间。关于选项，Vim通常试着给它们类似Unix的缩写；若是命令，则以最短的独特初始子字符串作为缩写。

一些常用命令的缩写：

```
n       next
prev    previous
q       quit
se      set
w       write
```

一些常用选项的缩写：

```
ai      autoindent
bg      background
ff      fileformat
ft      filetype
ic      ignorecase
li      list
nu      number
sc      showcommand（不是showcase）
sm      showmatch
sw      shiftwidth
wm      wrapmargin
```

熟知命令与选项的时候，它们的缩写能节省时间。但在.vimrc或gvimrc文件中编写脚

本以及使用命令设置会话时，采用完整命令与选项名称反而能节省时间（就长期效益而言）。因为使用完整名称，你的配置文件与脚本将较容易理解，也较容易调试。

注意： 请注意，这不是在Vim发布版本中的整套脚本（syntax、autoindent、colorscheme等等）采用的方式，不过我们对于接受默认的方式没有问题。这里只是推荐，为了轻松管理自己的脚本，最好使用完整名称。

几项快捷窍门（不只Vim专用）

现在我们提供几项技巧——有些是vi与Vim都提供的，它们都很值得记忆并设置得容易使用：

快速交换字符

输入时误把两个字符的位置颠倒是项很常见的拼写错误。把光标放在第一个顺序错误的字符上，然后输入xp即可（剪切字符、放置字符）。

另一种快速交换字符

想交换两行的位置吗？把光标放在上一行，然后输入ddp（删除一行、放在当前行的上方）。

快速调用帮助

别忘了Vim的内置帮助。单击功能键F1，就能分割屏幕并显示在线帮助的介绍。

我使用的命令是什么？

以最简单的形式，Vim让我们在命令行上用箭头取用最近执行的命令。请使用上下箭头调用命令，Vim可显示最近使用过的命令，而且可以编辑。无论是否编辑Vim历史记录中的命令，均可按下ENTER键执行该命令。

调用Vim的内置的命令历史记录编辑，则可执行更复杂的操作。请于命令行输入CTRL-F。这将打开一个小“命令”窗口（默认高度为7），在其中能使用一般的Vim移动命令，并可以像在正常Vim缓冲区中般搜索并改变内容。

在命令编辑窗口中，可以轻易找到最近的命令，如果需要就修改，并按下ENTER键执行。亦可选择在缓冲区写入某个文件，记录命令历史以备未来所需。

一点幽默感

请试着输入下列命令：

```
:help sure
```

然后阅读Vim的响应。

参考资源

这里有两个链接，以HTML呈现Vim的内置帮助（Vim最新的两次重大发布）：

Vim 6.2

http://www.vim.org/htmldoc/help.html

Vim 7

http://vimdoc.sourceforge.net/htmldoc/usr_toc.html

除此之外，*http://vimdoc.sourceforge.net/vimfaq.html*收集了与Vim相关的最常见问题。它并没有问题与答案间的超链接，但都在同一页面。可以试着向下滚动找到答案，并开始扫读需要的内容。

Vim的官方网页以前曾经有关于Vim相关技巧的信息，但因为垃圾邮件的问题，系统管理员把相关信息移到wiki上（比较容易处理垃圾邮件的问题）。wiki的地址是*http://vim.wikia.com/wiki/Category:Integration*。

第三部分

其他vi同类品

第三部分讨论其他受欢迎的、与Vim同时成长的vi同类品。该部分包含下列章节：



第十六章

nvi：新的vi

nvi 是“新的vi”（new vi）的简写。它的发源地是加州大学伯克利分校（University of California at Berkeley, UCB），也是有名的BSD（Berkeley Software Distribution）版本的 Unix 的发源地。本章就是使用nvi编写的。

作者与历史

最初的vi开发于20世纪70年代晚期。作者是Bill Joy，当时是计算机科学系的研究生，现在是Sun Microsystems的创始人与副总裁。

Bill Joy原本在开发ex（由第六版的ed编辑器做大幅的改进而来）。第一个改进是开放模式，由Chuck Haley完成。在1976年到1979年之间，ex演变成vi。接着Mark Horton来到伯克利，加入了宏与“其他特色”（注1），并做了许多让vi能够在各种终端与Unix系统上执行所需的工作。在4.1BSD（1981年）时，vi编辑器基本上已经拥有本书第一部分描述的所有特色了。

尽管改变了不少，但vi的核心代码是原始Unix ed编辑器。因此，它是不能自由发布的代码。在20世纪90年代早期，BSD的开发者正在开发4.4BSD时，他们想要一种可以自由发布源代码的vi版本。

UCB的Keith Bostic从elvis 1.8（注2）这种可以自由发布的vi同类品开始，将其修改成与vi“错误兼容”（bug for bug compatible）的同类品。nvi也与POSIX Command Language和Utilities Standard（IEEE P1003.2）兼容，这是很合理的。

注1：这句出自nvi的参考手册。很可惜，手册中没说明是哪些特色。

注2：虽然最初的elvis源代码已经几乎找不到了。

虽然与UCB不再有关系，但Keith Bostic仍继续维护、改进并发布nvi。本书编写时的版本是nvi 1.79。

nvi的重要性在于它是“正宗”伯克利版本的vi。它是 4.4BSD-Lite II的一部分，而且是各种常见的BSD，如NetBSD与FreeBSD中使用的vi。

重要的命令行参数

在纯粹的BSD环境中，nvi安装在ex、vi与view的名称下。一般来说，它们都会链接到同一个可执行文件，nvi则以打开的方式来决定其行为（Unix中的vi也会这样做）。nvi允许用Q命令从vi模式转换到ex模式。而view除了初始化时设置readonly选项外，其他都与vi相同。

nvi有许多命令行参数。本处介绍最有用的的几个参数：

-c *command*

在启动时执行*command*。这是旧的+command语法的POSIX版本，但是nvi并不限制命令的位置（旧的语法也可使用）。

-F

在开始编辑时不复制整个文件。如此可能会加快速度，但是别人有可能在你编辑文件时将文件改变。

-r

恢复指定的文件；如果命令行没有列出文件名，就列出所有可以恢复的文件。

-R

以只读模式启动，设置readonly选项。

-s

进入批（脚本）模式。这只针对ex，作为编辑脚本之用。提示符与非错误消息不会显示。这是过去的“-”参数的POSIX版本，nvi对两种版本都支持。

-S

运行时设置secure选项，不允许访问外部程序（注3）。

-t *tag*

从指定的*tag*处开始编辑。

注3： 与其他标有“安全”（secure）的事物一样，盲从安全标志通常都不甚合适。不过，Keith Bostic说，我们可以信任nvi的secure选项。

-w *size*

将初始窗口尺寸设置成*size*行。

在线帮助与其他说明文档

nvi拥有容易理解又可打印的说明文档。特别是它包含了troff源代码、格式化的 ASCII 与格式化的PostScript版本的文件。内容包括：

vi参考手册

nvi的参考手册。手册中说明nvi的命令行选项、命令、选项与ex命令。

vi手册页

nvi的手册页。

vi使用教学

介绍如何用vi编辑的使用教学。

ex参考手册

ex的参考手册。这份手册是针对最初的ex所写，对现在的nvi功能来说，已经有点过时了。

另外还包括一些ASCII文件，有的记录了一些nvi的内部信息，有的提供应该要实现的功能，同时还包括vi的在线教学。

nvi内置的在线帮助并不多，由两个命令组成，:exusage与:viusage。它们分别为每一个ex与vi命令提供单行说明。如果只是想复习某个命令的用途，这就足够了，但是如果要学习不常见或新的nvi功能，这就不适合了。

若以命令名称作为:exusage与:viusage命令的参数，nvi将只显示这个命令的在线帮助消息。nvi会用一行解释这个命令的功能，再用另一行总结命令的用法。

初始化

如果设置了-s或“-”选项，则nvi会跳过所有的初始化动作；否则，nvi会依序执行下列步骤：

1. 读入并执行/etc/vi.exrc文件。该文件的拥有者必须是root或是你自己。
2. 如果有环境变量NEXINIT，则予以执行；否则，执行现有的EXINIT。两者中只有一个会被使用，不会同时执行。此时会跳过$HOME/.nexrc或$HOME/.exrc。

3. 如果存在$HOME/.nexrc，则读入并执行；否则，查找$HOME/.exrc是否存在，读入并执行。两者中只有一个会被用到。

4. 如果设置了exrc选项，则会查找./.nexrc或./.exrc是否存在，若找到则予以执行。只会使用其中一个。

除非（可写入的）文件为用户所拥有，否则nvi不会执行。

nvi说明文档中推荐把共同的初始化动作放入你的.exrc文件（也用于Unix vi的选项与命令），而在.nexrc文件中执行:source .exrc，并将这一行放在nvi专用的初始化动作之前或之后。

多窗口编辑

要在nvi中创建新窗口，可以使用以下ex编辑命令：Edit、Fg、Next、Previous、Tag、Visual（这些命令都可以简写）。如果你的光标位于屏幕的上半部，新窗口会创建在下半部，反之亦然。接着可以用CTRL-W切换到另一个窗口：

```
<preface id="VI6-CH-0">
<title>Preface </title>

<para>
Text editing is one of the most common uses of any computer system, and
<command>vi</command> is one of the most useful standard text editors
on your system.
With <command>vi</command> you can create new files, or edit any existing
Unix text file.
</para>
choo.sgm:  unmodifed:  line 1

# Makefile for vi book
#
# Arnold Robbins

CHAPTERS = ch00_6.sgm ch00_5.sgm ch00.sgm ch01.sgm ch02.sgm ch03.sgm \
        ch04.sgm ch05.sgm ch06.sgm ch07.sgm ch08.sgm
APPENDICES = appa.sgm appb.sgm appc.sgm appd.sgm

POSTSCRIPT = ch00_6.ps ch00_5.ps ch00.ps ch01.ps ch02.ps ch03.ps \
        ch04.ps ch05.ps ch06.ps ch07.ps ch08.ps \
Makefile:  unmodifed:  line 1
```

上例显示了nvi编辑的两个文件：ch00.sgm与Makefile。分割窗口是输入nvi ch00.sgm再加上:Edit Makefile的结果。每一个窗口的最后一行作为状态行，即为该窗口执行冒号命令的地方。状态行采用高亮显示。

与窗口有关的ex模式命令与用途，均于表16-1中描述。

表16-1：nvi的窗口管理命令

命令	作用
bg	隐藏当前的窗口。之后可以用fg或Fg命令重新调用
di[splay] b[uffers]	显示所有的缓冲区，包括命名的、未命名的与编号的缓冲区
di[splay] s[creens]	显示所有后台窗口对应的文件名
Edit *filename*	在新的窗口中编辑*filename*
Edit /tmp	创建新窗口以编辑空的缓冲区。/tmp被特别解释成创建新的临时文件
fg *filename*	在当前窗口中显示*filename*。之前的文件则移到后台
Fg *filename*	在新的窗口中显示*filename*。当前窗口会被分割，而不是将屏幕空间重新分配给所有打开的窗口
Next	在新的窗口中编辑参数列表中的下一个文件
Previous	在新的窗口中编辑参数列表中的前一个文件（nvi中具有相应的previous命令，可移动到前一个文件，但Unix的vi中没有这个命令）
resize +-*nrows*	将当前窗口的行数增加或减少*nrows*行
Tag *tagstring*	在新窗口中编辑包含*tagstring*的文件

CTRL-W命令会在窗口间从上往下循环。用:q与ZZ命令可离开当前的窗口。

你可以在多个窗口中打开同一个文件。在一个窗口中做的改变会反映在其他的窗口中，然而在nvi的插入模式中所做的改变需等到按下ESC结束编辑后，才会在其他窗口中出现。除非执行一个让nvi离开某个文件的最后一个窗口的命令，否则nvi不会出现保存文件改变的提示。

图形用户界面

nvi并未提供图形用户界面的版本。

扩展正则表达式

扩展正则表达式已于第131页的“扩展正则表达式”一节中介绍。本节只是整理nvi提

供的元字符（metacharacter）。nvi也支持POSIX的方括号表示法，例如[[:alnum:]]等等。

使用:set extended可打开扩展正则表达式匹配：

|
    指示交替。左右两边不必是单一的字符。

(...)
    用于分组，可让其他正则表达式运算符来使用。

    设置extended后，以括号括起的文本的行为与一般vi中使用\(...\)括起的文本一样；实际匹配到的文本即是可以在替换命令的替换部分中使用\1、\2等等返回。在这里，\(表示一个字面的左括号。

+
    +前面的正则表达式需匹配一次或多次。可以匹配单一字符或一组由括号括起的字符。

?
    ?前面的正则表达式需匹配零次或一次。

{...}
    定义一个区间表达式（interval expression），以指定重复的次数。下面提到的*n*与*m*代表整数：

    {*n*}
        前面的正则表达式恰好匹配*n*次。

    {*n*,}
        前面的正则表达式匹配*n*次以上。

    {*n*,*m*}
        前面的正则表达式被匹配的次数介于*n*到*m*次。

    未设置extended时，nvi则提供相同于\{和\}的功能。

或许有人已经想到了，如果设置了extended，你需要在上面的元字符前加上反斜线，这样匹配时才会把它们当成实际的字符。

改进的编辑功能

本节说明nvi让简单的文本编辑更容易、更强大功能。

命令行历史记录与自动补全

nvi会保存你的ex命令行，并可以将其修改后重新执行。

这个功能由cedit选项控制，选项值为一个字符串。

当我们在冒号命令行中输入第一个字符，nvi即打开一个命令历史记录的新窗口，以备稍后编辑所需。当你在命令历史记录窗口的某一行按下ENTER，nvi会执行那一行的命令。ESC是这个选项的一个较好选择（使用^V ^[输入）。

由于ENTER键会实际执行命令，在使用j或↓键往下移动一行时，请小心。

除了编辑命令行，你也可以实现文件名的展开。这个功能由filec选项控制。

当你在冒号命令行中输入这个字符串的第一个字符时，nvi会将这个字符之后的空格当成*（通配符），之后以shell样式将文件名展开。ESC也是这个选项的一个较好选择（用^V ^[输入）。当这个字符与cedit选项相同时，只有在它是冒号命令行的第一个字符时，才会执行命令行编辑。

注意： nvi说明文档提到TAB也是filec选项的常见选择。想让它运作，必须输入:set filec=\TAB。实际上，使用ESC作为两个选项已经运作得不错了。

在你的.nexrc文件中设置这些选项是最容易的：

```
set cedit=^[
set filec=^[
```

标签栈

标签堆栈已在第134页的“标签栈”一节中介绍。4种同类品中，nvi的标签栈最简单。表16-2与表16-3显示了它所使用的命令。

表16-2：nvi的标签命令

命令	功能
di[splay] t[ags]	显示标签栈
ta[g][!] *tagstring*	编辑包含*tagstring*并已定义在tags文件中的文件。如果当前的缓冲区被修改过，但是还没保存，!会强迫nvi切换到新文件
Ta[g][!] *tagstring*	功能就像:tag，但在新的窗口中做编辑

表16-2：nvi的标签命令（续）

命令	功能
tagp[op][!] *tagloc*	弹出指定的标签。如果没有指定*tagloc*，则弹出最近一次使用的标签。*tagloc*位置可以是某个标签的文件名称，或是指示栈中某个位置的数值
tagt[op][!]	弹出栈中最旧的标签并在进程中清除栈

表16-3：nvi的命令模式标签命令

命令	功能
^]	在tags文件中寻找光标下的标识符位置并移到那个位置。当前位置会自动压入标签栈
^T	回到标签栈中的前一个位置，也就是弹出一个元素

你可以将tags选项设置为nvi应该查找标签的文件名列表。这提供了一种极简便的搜索路径机制。默认值是"tags /var/db/libc.tags /sys/kern/tags"，若在4.4BSD系统上，其将先查看当前目录，再查看C库与操作系统源代码中的文件。

taglength选项会控制标签字符串中有多少字符是有效的。默认值为0，表示使用所有的字符。

nvi有一点与vi一样："单词"从光标所在位置开始计算。如果光标位于*main*中的*i*上，nvi会搜索*in*，而不是*main*。

nvi依靠传统的tags文件格式。但这种格式的功能有限，尤其是缺少编程语言中的作用域（scope）的概念，而作用域可让同一个标识符在不同上下文中表示不同的意义。这个问题在C++中特别严重，因为C++允许函数名称重载（overloading），也就是说，可以用同一个名称表示不同的函数。

nvi用另一种完全不同的方法回避tags文件的限制：cscope程序。cscope虽非免费，却很便宜，可至Bell Labs Software Toolchest取得（请参考*http://www.bell-labs.com/project/wwexptools/*）。它会读入C的源文件并构建一个描述程序的数据库。nvi提供查询数据库的命令，让我们能处理结果。因为cscope不是随处都找得到，故我们在这里并不介绍用法。nvi相关命令的细节请查阅nvi的说明文档。

Exuberant ctags产生的扩展标签文件格式在nvi 1.79中不会产生任何错误，但是，nvi也不能利用这种格式的优点。

不限次数的撤销

在vi中，点号（.）命令通常表示“再做一次”，它会重复上一次的编辑动作，但需为删除、插入或替换动作。

nvi将点号命令做了一般化，使之成为完整的“重做”命令，即使上一个命令是 u（撤销）命令，也一样可用。

因此，要打开一连串的“撤销”命令，首先输入u。接着，每按一次.，nvi将撤销一次改变，逐渐把文件恢复到接近未编辑前的原始状态。

最后，会回到文件的初始状态。此时，再按下.只会发出蜂鸣声（或屏幕闪动）。你现在可以执行u来“撤销上一次的撤销”，接着再用.一步步重新执行原来所做的改变。

nvi不允许在u或.命令中加上次数。

任意长度的行与二进制数据

nvi可以编辑包含任意长度的行以及任意数量的行的文件。

nvi会自动处理二进制数据，不需要使用特殊的命令行选项或ex选项。可以用^X加上一个或两个十六进制数字，以便在文件中输入8位的字符。

增量搜索

在nvi中，使用:set searchincr启用增量搜索。

输入时，光标会在文件中移动，永远位于匹配文本的第一个字符上。

左右滚动

在nvi中，使用:set leftright启用左右滚动。sidescroll的值会控制nvi往右滚动时屏幕所移动的字符数。

编程辅助

nvi并不支持特定编程辅助功能。

有趣的功能

nvi是各种同类品中功能最少的一个，就算有什么新增功能，我们也都在其他同类品上看过了。然而还是有一些重要的特色值得一提。

国际化支持

nvi中大部分的信息与警告消息都可以使用一种称为“消息目录”（message catalog）的功能，翻译为其他语言。nvi实现了这个功能，并使用在nvi发行包中的 catalog README文件中说明的直接方法。消息目录提供了荷兰文、英文、法文、德文、俄文、西班牙文与瑞典文版本。

任意的缓冲区名称

历史上，vi的缓冲区名称只限使用26个英文字母。nvi让你能够使用任何字符作为缓冲区名称。

对 /tmp 的特殊解释

对于任何需要用文件名作为参数的ex命令，如果你使用/tmp这个特殊名称，nvi会将其替换成某个唯一的临时文件名。

资源与支持的操作系统

nvi可至*http://www.bostic.com/vi*取得。从这个网站可以下载最新的版本（注4），你也可以加入他们的电子邮件列表，这样在nvi发表新版本或新功能时会收到通知。

nvi的源代码可以自由取得。许可条款可在发行包中的LICENSE文件中找到，以源程序与二进制文件的形式来发布是被允许的。

nvi可以在Unix下构建并运行。它也可以在LynxOS 2.4.0上运行，或许亦可在其后的版本上运行。nvi还可能在其他符合POSIX规范的系统上构建并运行，但是在它的说明文档中并未列出nvi支持的操作系统。

编译nvi的过程非常简单。通过ftp取得发行包，将其解压缩并逐个打开文件之后，运行configure程序，接下来运行make：

```
$ gzip -d < nvi.tar.gz | tar -xvpf -
```

注4： 目前正在开发GUI版的nvi，有兴趣的读者请参考网站上的联络信息。

```
...
$ cd nvi-1.79; ./configure
...
$ make
...
```

nvi应该会顺利配置并编译。然后可用make install来安装。

如果你需要报告nvi的错误或问题，可以联络Keith Bostic，他的电子邮件地址为：*bostic@bostic.com*。



第十七章

Elvis

elvis由Steve Kirkendall编写与维护。其早期的版本后来成为nvi的基础。本章即使用elvis 编写。

作者与历史

为了表达我们的感谢之意，就让Steve Kirkendall用自己的话来叙述elvis的历史：

> 以前使用一个名为stevie的早期同类品时，这个程序的死机害我损失了好几个钟头的工作结果，并完全摧毁了我对这个程序的信心，我便开始着手设计elvis 1.0。另外，stevie将编辑缓冲区存储在RAM中，这对于Minix实在不怎么实际。因此我开始编写自己的同类品，将编辑缓冲区存储在文件中。即使我的编辑器崩溃了，我还是可以从文件中取回已编辑的文本。
>
> elvis 2.x与1.x几乎完全不同。为什么我要写出第二个vi同类品呢？因为第一个背负了太多原始vi与Minix的包袱。第二个最大的改变是支持多个编辑缓冲区与编辑窗口，这些都不是可以在1.x中轻易加入的功能。我也希望不再限制一行的长度，并用HTML来写在线帮助。

对于使用“elvis”这个名字，Steve说其中至少有部分的原因是想看看有多少人会问他为什么会取这个名字（注1）！对vi的同类品来说，在名称中有“vi”这两个字母，也是很常见的事。

重要的命令行参数

elvis一般并不会安装成vi，虽然也可以这样做。如果以ex来调用，则它会成为行编辑器并可以用Q命令从vi模式切换到ex模式。

注1：　大约8年前，我是第四个问这个问题的人！——A. R.

elvis有许多命令行参数。下面介绍其中一些最有用的参数：

-a

将命令行指定的每一个文件载入独立的窗口中。

-c *command*

在启动时执行*command*。这是过去 +command语法的POSIX版本（旧的语法也可使用）。

-f *filename*

使用*filename*作为会话文件（session file），代替默认的名称。稍后另有关于会话文件的讨论。

-G *gui*

使用指定的界面。默认采用termcap界面。其他选择包括x11、win32、curses open与quit。你所用的elvis版本不一定会把所有的界面都编译进去。

-i

以输入模式开始编辑，而不是命令模式。这对新手来说可能比较容易。

-o *logfile*

将启动消息重定向至指定文件，而不是stdout/stderr。这个参数对MS Windows用户非常重要，因为Windows放弃任何写入标准输出（standard output）与标准错误（standard error）的内容，这将使WinElvis的配置问题几乎不能侦测。有了-o *logfile*，我们可以把诊断信息发送到某个文件，以备日后查看。

-r

在程序死掉后执行恢复动作。

-R

以只读模式开始编辑文件。

-s

从标准输入（standard input）中读入并执行ex脚本（依照POSIX标准）。这个参数绕过了所有初始化脚本。

-S

为整个会话设置选项security=safer，不只执行.exrc文件。这个参数增加了某种程度的安全性，但不应盲目地信任。

-SS

设置选项security=restricted，比security=safer更疑神疑鬼的设置。

-t tag

从指定的 *tag* 处开始编辑。

-V

输出更详细的状态信息。这在诊断初始化文件的问题时会有用。

-?

打印可能的选项摘要。

在线帮助与其他说明文档

elvis在这方面很有趣。在线帮助非常具有启发性，并完全使用HTML编写，方便大家用自己最喜欢的网页浏览器观看。elvis也有HTML的显示模式（在后面介绍），这样在elvis中观看在线帮助就非常容易并且愉快。

在观看HTML文件时，可以使用标签命令（^]与^T）跳到不同的主题再返回，这使浏览帮助文件变得很容易。我们十分赞赏这项创新。

当然，elvis也附有Unix的手册页。

初始化

这一节描述elvis的会话文件，并分项列出初始化步骤。

会话文件

elvis的目标是符合“通用开放系统环境”（Common Open System Environment, COSE）的标准。这个标准要求程序能够保存它的状态，使得之后可以回到这个保存的状态。

为了做到这一点，elvis在会话文件中维护所有的状态。正常来说，elvis会在启动时创建会话文件，而在结束时删除，但是如果elvis崩溃了，遗留的会话文件即可用于已恢复编辑的文件。

初始化步骤

elvis会执行以下的初始化步骤，有趣的是，elvis大部分的自定义功能都从编辑器的选项移到了初始化文件中：

1. 初始化所有固定的选项。

2. 从已编译在elvis中的界面中选择一种。elvis会选择其中“最好的”一个。例如，X11界面就被认为比termcap界面好，但是如果没有运行X Windows，就不能使用。所选择的界面可以处理命令行中针对此界面的初始化选项。
3. 如果不存在会话文件的话，则创建此文件；若有，则读入（作为恢复之用）。
4. 在ELVISPATH环境变量中初始化elvispath选项，否则就将默认值赋予变量。通常的值是"~/.elvislib:/usr/local/lib/elvis"，但实际的值会因elvis的配置与编译而有不同。
5. 从elvispath中寻找一个名为elvis.ini的ex脚本并运行。默认的elvis.ini文件会执行下列动作：
    - 依照当前的操作系统选择一个digraph表（digraph用于定义系统的扩展 ASCII 字符集以及如何将其输入）。
    - 依照程序的名称设置选项（例如，ex或vi模式）。
    - 处理与系统相关的小细节，例如设置X11的颜色以及在界面上增加菜单。
    - 选择一个初始化的文件名，在Unix中是.exrc，而在非Unix系统中可能是elvis.rc。我们称这个文件为f。
    - 如果存在EXINIT环境变量，就执行其值；否则，执行:source ~/f，其中f就是先前选择的文件。
    - 如果设置了exrc选项，则在当前目录的f文件上执行safely source命令。
    - 在X11中，如果没有设置字体的话，则设置字体为正常、黑体与斜体。
6. 载入读入前、读入后、写入前与写入后的命令文件（如果有这些文件的话）。elvis.msg文件也要载入。本章稍后将另外介绍这些文件。
7. 载入并显示命令行中的第一个文件。
8. 如果给定-a选项，则载入其他的文件，并在独立的窗口中显示。

多窗口编辑

要在elvis中创建新的窗口，则先使用ex的:split命令。接着使用一般的ex命令，如:e *filename*或:n来编辑新文件。这是最简单的方法。本章稍后另外介绍其他更短的方法。你可以用CTRL-W CTRL-W在窗口间切换：

```
<preface id="VI6-CH-0">
<title>Preface </title>

<para>
Text editing is one of the most common uses of any computer system, and
<command>vi</command> is one of the most useful standard text editors
on your system.
With <command>vi</command> you can create new files, or edit any
existing Unix text file.
________________________________________________________________________
# Makefile for vi book
#
# Arnold Robbins

CHAPTERS = ch00_6.sgm ch00_5.sgm ch00.sgm ch01.sgm ch02.sgm ch03.sgm \
        ch04.sgm ch05.sgm ch06.sgm ch07.sgm ch08.sgm
APPENDICES = appa.sgm appb.sgm appc.sgm appd.sgm

POSTSCRIPT = ch00_6.ps ch00_5.ps ch00.ps ch01.ps ch02.ps ch03.ps \
        ch04.ps ch05.ps ch06.ps ch07.ps ch08.ps \
        appa.ps appb.ps appc.ps appd.ps
```

这个分割画面是输入elvis ch00.sgm加上:split Makefile的结果。

像nvi一样，elvis会给每一个窗口添加独立的状态行。elvis的独特之处在于，使用一行强调用的下划线当作状态行，而不是高亮显示。ex的冒号命令会在每一个窗口的状态行中出现。

表17-1描述了与窗口有关的ex模式命令及其用途。

表17-1：elvis的窗口管理命令

命令	功能
sp[*lit*] [*file*]	创建新窗口，如果有*file*参数，则会载入此文件；否则，以新窗口显示当前的文件
new sne[w]	创建一个新的空缓冲区，接着创建一个新的窗口来显示这个缓冲区
sn[ext][*file*...]	创建一个新窗口，显示参数列表中的下一个文件。当前的文件不受影响
sN[ext]	创建一个新窗口，显示参数列表中的前一个文件。当前的文件不受影响
sre[wind][!]	创建一个新窗口，显示参数列表中的第一个文件。将“当前的”文件复位为相对于:next命令的第一个文件。当前的文件不受影响
sl[ast]	创建一个新窗口，显示参数列表中的最后一个文件。当前的文件不受影响

表17-1：elvis的窗口管理命令（续）

命令	功能
sta[g][!] *tag*	创建一个新窗口，显示发现*tag*的文件
sa[ll]	为任何位于参数列表中却没有对应窗口的文件创建新窗口
wi[ndow][*target*]	如果没有*target*，则列出所有的窗口。*target*的可能值于表17-2中描述
close	关闭当前的窗口。其中显示的缓冲区不会受影响。如果此缓冲区被修改过，则其他离开elvis的命令不会成功，除非显式保存或放弃缓冲区
wquit	将缓冲区写回文件并关闭窗口。不管文件有没有被修改，都会被保存
qall	对每一个窗口下达:q命令。没有对应窗口的缓冲区不会受影响

表17-2描述了与窗口有关的ex参数与其意义。

表17-2：elvis 窗口命令的参数

引数	意义
+	切换到下一个窗口，如同^W k
++	切换到下一个窗口，如同^W ^W 一样绕回
-	切换到前一个窗口，如同^W j
--	切换到前一个窗口，最后将绕回末端
num	切换到window=*num*的窗口
buffer-*name*	切换到编辑命名缓冲区的窗口

elvis提供了许多vi模式的窗口间移动的命令，汇总于表17-3。

表17-3：vi命令模式中的elvis窗口命令

命令	功能
^W c	隐藏缓冲区并关闭窗口。与:close完全相同
^W d	在syntax模式与其他显示模式（html、man、tex）间切换显示模式。这使编辑网页较为方便。它是只用于个别窗口的选项。显示模式会在第340页的“显示模式”一节中讨论
^W j	往下移动到下一个窗口
^W k	往上移动到上一个窗口
^W n	创建一个新窗口，并创建一个新缓冲区显示在窗口中。与:snew命令类似

表17-3：vi命令模式中的elvis窗口命令（续）

命令	功能
^W q	保存缓冲区并关闭窗口，与ZZ相同
^W s	分割当前的窗口，与:split相同
^W S	切换wrap选项。用于控制过长的行是绕排至下一行，还是将整个屏幕往右滚动。它是只用于个别窗口的选项，将于第335页的“左右滚动”一节中讨论
^W]	创建一个新窗口，接着寻找光标所在位置的标签。与 :stag 命令类似
[count] ^W ^W	移到下一个窗口，或是第count个窗口
^W +	增加当前窗口的尺寸（只用于termcap界面）
^W -	减少当前窗口的尺寸（只用于termcap界面）
^W \	将当前窗口放到最大（只用于termcap界面）

图形用户界面

这一节的屏幕截图与解释是由Steve Kirkendall提供的，我们非常感谢。

elvis的X11界面提供了滚动条与鼠标支持，并可选择使用的字体。在elvis创建第一个窗口后就不能更改字体了。字体必须全部都是等宽（monospace）字体，通常是Courier或“定宽”字体的一些变形。

elvis的X11界面支持多种字体与颜色、通过改变形状来表示编辑模式（插入或命令）的闪烁光标、滚动条、反锯齿文本、用于背景的图像文件（可选择是否使用）、用户专用的图标以及鼠标动作。鼠标能用于选择文本、在应用程序间剪切及粘贴内容并可搜索标签。除此之外，还有可调整配置的工具栏、对话框、状态条、-client标志。

注意： MS Windows GUI也支持背景图像文件，请使用相同的命令并使用XPM格式文件，以便同一背景图像文件能用于两种环境中。

基本窗口

基本的elvis窗口如图17-1所示。

elvis另外提供弹出式的文本搜索对话框，如图17-2所示。

它们看起来都很像Motif，但是elvis实际上并没有使用Motif库。

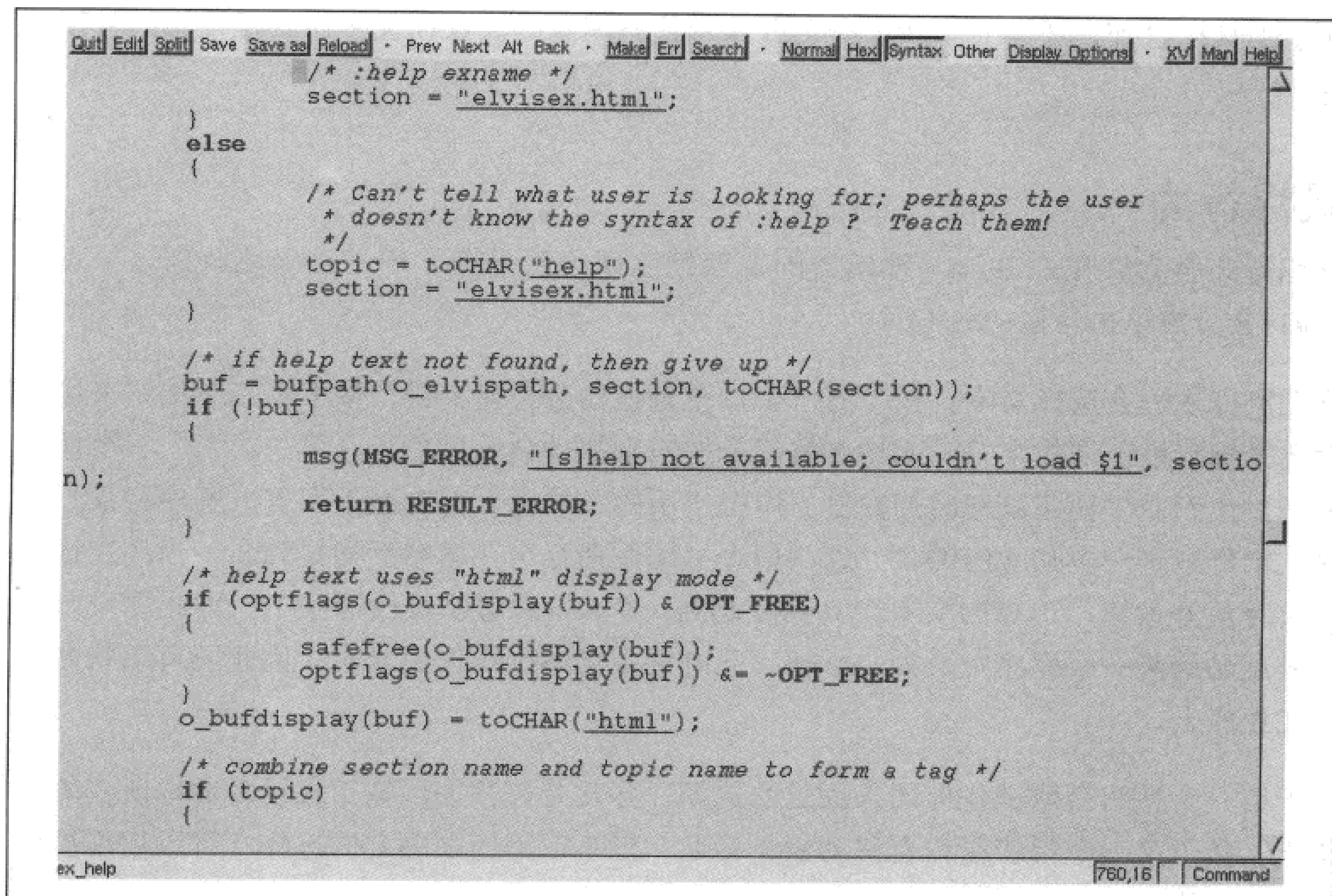


图17-1：elvis GUI窗口

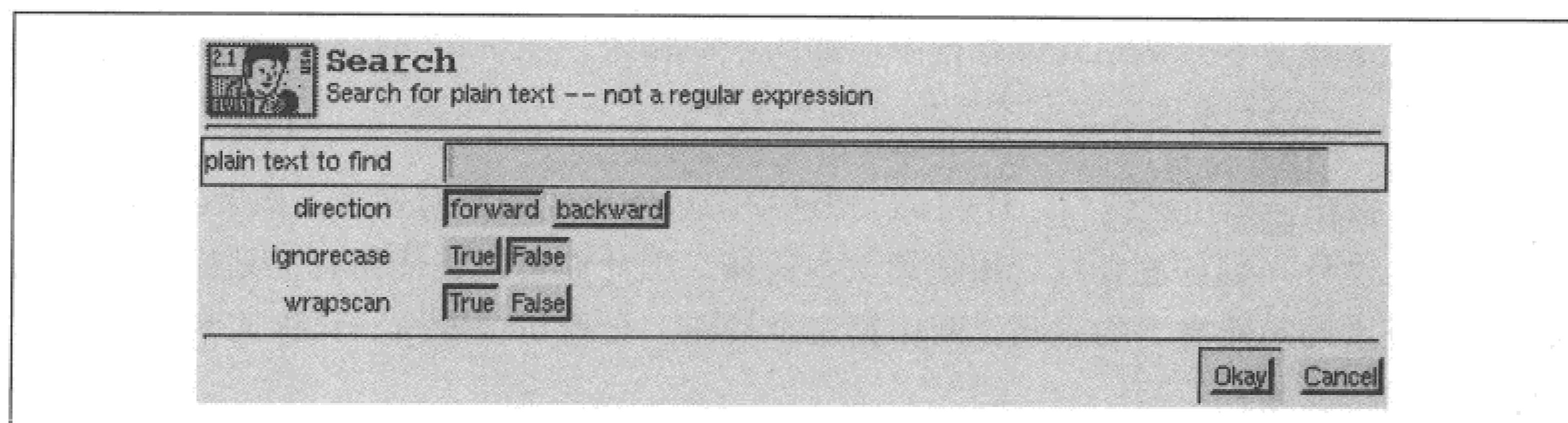


图17-2：elvis搜索对话框

命令行选项让我们可选择4种字体供elvis使用，包括正常、斜体、黑体与“控制”，控制字体是用于工具栏文字与按钮标签的字体。也可以指定前景与背景的颜色、初始窗口的几何形状以及elvis是否该在启动时缩成图标。

新的-client选项会让elvis查找已存在的elvis进程，并发送一个消息给后者，要求它开始编辑命令行上指定的文件。这样做可以在elvis当前编辑的文件与新文件之间共享拖动选中的文本与其他信息。

除了工具栏之外，还有一个状态条可以显示状态消息以及任何关于工具栏按钮的可用信息。

鼠标行为

鼠标行为会尝试在xterm与编辑器的合理性之间达到平衡。为了正确地做到这一点，elvis会分辨单击与拖动的不同。

拖动鼠标永远会选择文本。按下第一个按钮来拖动会选择字符，按下第二个按钮来拖动会选择矩形框区域的字符，而按下第三个按钮来拖动会选择整行（第一、二、三个按钮分别对应到右手使用鼠标时的左键、中键／滚轮、右键。对左撇子而言，顺序相反）。这些动作分别对应elvis的v、^V、V命令（这些命令会在本章后面提到）。当你在拖动结束时放开按钮，选择的文本会立刻复制到一个X11的剪贴板缓冲区中，因此你可以将内容粘贴到另一个程序中，如xterm。此时文本仍然维持选中状态，因此还可以将命令用在程序上。

按下第一个按钮会取消任何还未决定的选择，并将光标移动到按下按钮时鼠标所在的字符上。按下第三个按钮鼠标会移动到光标处，但不取消还未决定的选择，可以用来扩展还未决定的选择。

按下第二个按钮会“粘贴”X11剪贴板缓冲区中的文本（如同xterm）。如果正在输入ex命令行，则这些文本会粘贴到命令行中，如同手动输入。在粘贴的时候，按下鼠标的位置并不重要，elvis 永远会在文本光标处插入文本。

双击第一个按钮会模仿按键^]，让elvis把选中的单词当成标签而查找。如果elvis正在处理HTML文件，则标签查找会跟随超文本链接，因此你可以在任何有下划线的文本上双击，以查看描述此文本的主题。双击第三个按钮，会模仿按键^T，让你回到执行上一次标签查找的地方。

工具栏

X11界面支持用户自配置的工具栏。工具栏默认是启用的，除非你的~/.exrc中包含:set notoolbar命令。

默认的工具栏中已经定义了一些按钮了。使用:gui命令即可重新配置工具栏。

命令的数目很多。可以根据自己的喜好重新配置工具栏，包括：删除一个或所有现有的按钮、加入新的按钮、控制按钮的间距与按钮的分组等等。下面是一个简单的例子：

```
:gui Make:make
```

```
:gui Make " Rebuild the program
:gui Quit:q
:gui Quit?!modified
```

这些命令增加了两个新的按钮。第一行加入了一个名为Make的按钮，按下时会执行:make命令（:make命令会在本章后面介绍）。第二行加入了Make按钮的描述文字，在按下按钮时会出现在状态行中。此时，"并不是注释的开头，而是gui命令的一个运算符。

第二个按钮名为Quit，由第三行的命令创建。它会结束程序。第四行的命令则更改了Quit按钮的行为。如果条件（!modified）为真，按钮正常作用；但条件如果不成立，此按钮会忽略任何鼠标的按下动作，会显示成“扁平”的样子，而不是一般立体的样子。因此，如果当前文件被修改过，我们不能使用Quit按钮结束程序。

我们可以创建弹出式对话框，在按下特定工具栏按钮时出现。对话框可以设置预定义变量（选项）的值，而后可由与此按钮相关的ex命令负责测试。预定义变量有26个，名称为a～z，专门给此种用途的“程序”使用。下例结合了对话框与一个名为Split的新按钮：

```
:gui Split"Create a new window, showing a given file
:gui Split;"File to load:" (file) f = filename
:gui Split:split (f)
```

第一个命令将描述文字与Split按钮结合。第二个命令创建弹出式对话框，提示符号是File to load:并设置了filename选项。(file) 表示可以输入任何字符串，但是TAB键可以用来执行文件名的补全。f = filename会将filename的值复制到f中。最后，第三个命令实际执行:split命令，加上f的值，即是用户提供的新文件名。

这项功能非常具有灵活性，想知道完整的细节，可以参阅在线帮助文件。

选项

X11界面由非常多的选项所控制。一般是在.exrc文件中进行设置。其中包括各种选项与缩写以及各种字体、启用与自配置工具栏、状态条、滚动条与光标的设置等等。还有一些选项在用^W ^W切换窗口以及elvis结束回到原始的xterm时控制光标的行为。

在线说明文档描述了所有与X11相关的ex选项。我们介绍其中一些比较有趣的选项：

autoiconify

一般来说，当^W ^W命令将焦点移到一个成为图标的窗口时，这个窗口就会离开图标状态。当autoiconify设为ture时，elvis把旧的窗口变成图标，因此打开的elvis窗口会维持一定数量。

blinktime

此值介于1到10之间，表示光标从显示到不显示之间要经过多少个1/10秒。如果值为0，表示不闪烁。

firstx，firsty，stagger

firstx与firsty控制elvis创建的第一个窗口的位置。如果没有设置，则由-geometry选项或由窗口管理器来控制。如果stagger并非设置为零，则任何新的窗口会在当前窗口位置往右与往下移动该值的位置出现；如果设为零，则由窗口管理器决定位置。

stopshell

存储运行交互式shell的命令，即ex命令:shell与:stop以及可视命令^Z的行为。默认值为xterm &，它会在另一个窗口中打开一个交互式终端仿真器。

xscrollbar

其值若为left与right，可将窗口滚动条放在指定的一侧，而为none则禁用滚动条。默认值为right。

elvis可通过X资源来配置（注2）。资源中的值可被命令行中的标志取代，或在初始化脚本中使用:set或:color命令显式设置。表17-4中列出了elvis的资源。

表17-4：elvis 的X资源

资源类型	类型	默认值
elvis.Toolbar	Boolean	true
elvis.Statusbar	Boolean	true
elvis.Font	Font	fixed
elvis.Geometry	Geometry	80x34
elvis.Foreground	Color	black
elvis.Background	Color	gray90
elvis.MultiClickTimeout	Timeout	3
elvis.Control.Font	Font	variable
elvis.Cursor.Foreground	Color	red
elvis.Cursor.Selected	Color	red
elvis.Cursor.BlinkTime	Timeout	3

注2：X资源是配置X11应用程序的一种方式，根据X server存储于内存中的一个“名称／值”对集合设置。在当前的台式计算机环境，例如KDE与GNOME，已不常使用。但无论如何，仍然能使用xrdb命令予以设置。

表17-4：elvis 的X资源（续）

资源类型	类型	默认值
elvis.Tool.Foreground	Color	black
elvis.Tool.Background	Color	gray75
elvis.Scrollbar.Foreground	Color	gray75
elvis.Scrollbar.Background	Color	gray60
elvis.Scrollbar.Width	Number	11
elvis.Scrollbar.Repeat	Timeout	4
elvis.Scrollbar.Position	Edge	right

其中“Timeout”类型给予时间值，以1/10秒为单位。而“Edge”类型给予滚动条的位置，值可为left、right或none。

例如，如果你的X资源数据库中包含一行elvis.font:10x20，则默认的文本字体会是10x20。若未设置normalfont选项，则会使用此值。

扩展正则表达式

扩展的正则表达式已于第131页的“扩展正则表达式”一节介绍。其他在elvis中可用的元字符包括：

\|

表示交替（alternation）。

\(...\)

用于分组（grouping），允许额外运用正则表达式运算符。

\+

前面的正则表达式需匹配一次或多次。

\?

前面的正则表达式需匹配一次或零次



\{⋯\}

描述一个区间表达式，例如\{1,3\}可匹配*x*、*xx*或*xxx*。

亦可使用POSIX方括号表示法（例如字符类）。

改进的编辑功能

前一节描述了elvis让简单的文字编辑更容易的强大功能。

命令行历史记录与自动补全

你在ex命令行中输入的所有内容都会保存在名为Elvis ex history的缓冲区中。它可以像任意其他elvis缓冲区一样被访问，但是如果只是显示在窗口中，则并没有直接的用处。

要访问历史记录，可以使用方向键来显示前一个命令并对其做编辑。使用↑与↓在列表中移动，用←与→在命令行中移动。你可以通过键入插入字符或通过退格键删除字符。与在一般的vi缓冲区中编辑很像，按退格键会删除字符，但是行内容在你输入时并不会更新，请注意！

在Elvis ex history缓冲区中输入文本时（也就是在冒号命令行中输入），TAB键可以用于文件名称的扩展。前缀字符会被当成部分文件名，elvis会搜索所有匹配的文件。如果有多个匹配结果，则会尽可能匹配最多字符，然后发出蜂鸣声；若匹配的文件名中未隐含额外字符时，则elvis会列出所有匹配的名称并重新显示命令行。如果只找到单个的匹配名称，elvis即补全文件名称并加上一个tab字符。如果没有匹配的，则只会加上一个tab字符。

要获得真正的tab字符，请在输入前加上^V。也可以禁用文件名的自动补全，只要设置Elvis ex history缓冲区的inputtab选项为tab即可，命令如下：

```
:(elvis ex history)set inputtab=tab
```

标签栈

标签堆栈已于第134页的“标签栈”一节中说明。在elvis中，标签堆栈非常直接，如表17-5与表17-6所示。

表17-5：elvis的标签命令

命令	功能
ta[g][!] [*tagstring*]	编辑包含*tagstring*并已定义在tags中的文件。如果当前的缓冲区被修改过，但是还没保存，!会强迫elvis切换到新文件
stac[k]	显示当前的标签栈
po[p][!]	弹出栈中的一个光标位置，将光标恢复到前一个位置

表17-6：elvis命令模式的标签命令

命令	功能
^]	在tags文件中寻找光标所在处标识符的位置，并移动到那个位置。当前的位置会自动被压入标签栈
^T	回到标签栈中的前一个位置，也就是弹出一个元素

与传统vi的差别在于输入^]时，elvis会寻找包含光标位置在内的完整单词，而不只是光标位置往前的部分单词。

在HTML模式（稍后将于第340页的“显示模式”一节中讨论）中，所有命令的作用方式一样，除了:tag需要URL作为参数，而不是标签名称。URL不需要依赖tags文件，因此tags 文件在HTML模式中会被忽略。elvis支持file:、http:、ftp: URL，也能通过FTP写入。只需在elvis预期文件名的部分提供 URL。为了访问你在FTP站点上的账号（非匿名账号），URL中的目录名称必须以/~.elvis开始，这样才会读入你的/.~netrc 文件，以寻找正确的名称与口令。html显示模式能很好地运用这些功能！（network 函数也能在 Windows与OS/2上运作。）

许多:set选项都会影响elvis如何处理标签，如表17-7所示。

表17-7：elvis 的标签管理选项

选项	功能
taglength, tl	控制要寻找的标签中有效字符的数量。默认值为零，表示所有字符都是有效字符
tags，tagpath	这个选项值是用于寻找tags文件的目录或文件名列表。elvis会在其中的目录项中寻找名为tags的文件。列表中的项以冒号分隔（在DOS\Windows中则以分号分隔），以便让目录名称中包含空白。默认值只有tags，会在当前目录中寻找名为tags的文件。可以设置TAGPATH环境变量覆盖其值

表17-7：elvis 的标签管理选项（续）

选项	功能
tagstack	设为true时，elvis把每个位置放到标签栈中。利用:set notagstack可禁用标签堆栈

elvis（本书编写时正在测试阶段）支持第八章介绍的扩展tags文件格式。elvis拥有自己的ctags版本（称为elvtags，以避免与标准版相冲突）。这个版本能产生前述的增强格式。以下是它产生的一个特殊的!_TAG_行范例：

```
!_TAG_FILE_FORMAT       2       /supported features/
!_TAG_FILE_SORTED       1       /0=unsorted, 1=sorted/
!_TAG_PROGRAM_AUTHOR    Steve Kirkendall        /kirkenda@cs.pdx.edu/
!_TAG_PROGRAM_NAME      elvis Ctags     //
!_TAG_PROGRAM_URL       ftp://ftp.cs.pdx.edu/pub/elvis/README.html  //
!_TAG_PROGRAM_VERSION   2.1     //
```

在elvis中，每一个窗口都有自己的标签栈。

一般而言，elvis有些创新的标签行为。当读到过载的标签时，elvis将试图猜测我们寻找的目标并首先显示最接近的内容。如果拒绝了系统提供的内容（只需再次按下 ^]或再次输入:tag），它会显示下一个最接近的匹配内容，依此类推。系统也标注被你拒绝或接受的标签的属性，并利用这些尝试错误改善日后的搜索。

:tag命令的语法已被扩展，以允许你通过功能而不只是标签名称来搜索标签。有太多细节以致不能深入了解，可参阅在线帮助中描述标签用途的相关章节。

还有一个:browse命令，一次查找所有匹配的标签，并据此创建一个HTML表格。在此表格中，可以通过超链接通向任何想要的匹配标签。

最后则是tagprg选项，若设置了此选项，将弃用内置的标签搜索算法，而改为运行外部程序来执行搜索。

不限次数的撤销

使用elvis，在撤销（undo）与重做（redo）超过一次的改变前，必须先设置undolevels选项，以决定允许elvis“撤销”的层数。负数将禁止任何撤销（应该没什么用）。elvis的说明文档中警告，每一层的撤销大约使用6KB的会话文件（描述编辑会话的文件），因此可能会快速地消耗完磁盘空间。说明文档建议undolevels的设置值不要高于100，并“尽可能低于此值”。

设置undolevels为非零的值后，按照平常的方式输入文字。接着，每一次下达u命令，

即撤销一层改变。要重做（撤销上一次的撤销），则使用（好记的）^R（Ctrl+R）命令。

在elvis中，undolevels的默认值为零，其令elvis模仿Unix的vi。这个选项分别应用至各个编辑缓冲区。请参阅第322页的“初始化步骤”一节，其中描述了如何对你所编辑的每个文件做此设置。

一旦设置了undolevels，为u或^R命令加上数值，可指定撤销或重做改变的次数。

任意长度的行与二进制数据

elvis编辑的文件里，可包含任意长度的行以及任意数目的行。

在Unix下，elvis不会使用特别方式处理二进制文件。在其他的系统上，则使用elvis.brf 文件设置binary选项，这样可以避免换行符的转换问题。你可以用^X加上两个十六进制的数字，来输入8位的文本。使用hex显示模式是一个编辑二进制文件的好方法。稍后于第339页“有趣的功能”一节中，将介绍elvis.brf文件与hex显示模式。

左右滚动

在elvis中使用:set nowrap启用左右滚动。sidescroll的值控制elvis从左往右滚动时移动的字符数。^W S命令会切换该选项的值。

可视模式

elvis允许一次选择一个区域文本，包括一次选择一个字符、一次选择一行或是一个矩形的范围，可使用表17-8中的命令。

表17-8：elvis块模式的命令字符

命令	功能
v	开始选择文本区域，每次一个字符的模式
V	开始选择文本区域，每次一行的模式
^V	开始选择文本区域，矩形模式

elvis会强调（高亮显示）选择的文本。作选择时，只要使用一般的移动命令即可。下例显示一块矩形文本区域：

```
The 6th edition of <citetitle>Learning the vi Editor</citetitle>
brings the book into the late 1990&rsquo;s.
```

```
In particular, besides the &ldquo;original&rdquo; version of
<command>vi</command> that comes as a standard part of every Unix
system, there are now a numer of freely available &ldquo;clones&rdquo;
or work-alike editors.
```

elvis只允许对选择的文本区域做某些操作。有些操作只对整行文本有用，即使已选择了不是整行的文本区域也一样（参阅表17-9）。

表17-9：elvis块模式的操作

命令	操作
c，d，y	更改、删除或拖动文本。只有d命令可以正确地在矩形上运作
<，>，!	往左或往右移动文本以及过滤文本。只对包含已标记了区域的整行有作用

在使用d命令删除文本区域之后，屏幕上会显示如下：

```
The 6th edition of <citetitle>Learning the vi Editor</citetitle>
brings the 90&rsquo;s.
In particulo;original&rdquo; version of
<command>vi as a standard part of every Unix
system, thef freely available &ldquo;clones&rdquo;
or work-alike editors.
```

编程辅助

本节描述elvis的编程辅助性能。

编辑－编译的加速

elvis提供了在编写程序时不用离开编辑器的命令。我们可以重新编译一个文件、重新构建整个程序以及一次处理一个编译器的错误消息。这些elvis命令汇总在表17-10中。

表17-10：elvis的程序开发命令

命令	选项	功能
cc[!] [*args*]	ccprg	运行C编译器。适用于重新编译个别文件
mak[e][!] [*args*]	makeprg	重新编译所有需要重新编译的文件（通常是通过make）
er[rlist][!] [*file*]		移到下一个错误的位置

cc命令会重新编译一个源文件，可以在冒号命令行中予以运行。例如，如果你正在编辑hello.c，此时输入:cc，elvis就会帮你编译hello.c。

如果对:cc命令提供了额外参数，则这些参数会被传给C编译器。此时，必须提供所有的参数，包括文件名。

:cc命令会执行ccprg选项的文本，其默认值为 "cc ($1?$1:$2)"。elvis会将$2设为当前源文件的名称，而将$1设为给予:cc命令的参数。如果提供了参数，则ccprg会使用它们；否则，便只会将当前文件的名称传给系统的cc命令（当然，你可以修改ccprg，以符合你的习惯）。

同样地，:make命令会重新编译所有需要重新编译的文件。它会执行makeprg选项，默认值为 "make $1"。因此，你可以输入:make hello来只重新编译hello程序，或者只输入:make来重新编译所有的文件。

elvis会捕获编译或make时产生的结果，并查找类似文件名或行号的内容。若发现则予以处理，并移到第一个错误出现的地方。:errlist命令会依序移到下一个错误发生的位置。在移动到每一个错误的位置时，elvis会在状态行中显示对应的错误消息。

如果你为:errlist命令提供了*filename*参数，elvis会从这个文件载入一批错误消息并移到第一个错误发生的位置。

vi模式的*（星号）命令与:errlist的作用相同。当你有许多错误要处理时，用起来会比较方便。

最后，有一个真的很好用的特性是，elvis会跟踪文件的修改。当你添加或删除行时，elvis会跟踪，因此当你移到下一个错误时，会进入正确的行，而不一定是编译器的错误消息中的绝对行号。

语法高亮显示

要让elvis进行语法高亮显示，则使用:display syntax命令。这个命令是针对个别窗口的（另一种elvis显示模式会在第340页的“显示模式”一节中介绍）。

使用:color命令即可直接指定文本的外观。首先为不同文本类型赋予高亮显示。例如syntax 显示模式中的可能性包括：

comment
: 如何显示程序语言的注释

function

如何显示用于函数名称的标识符

keyword

如何显示用于关键字的标识符

prep

如何显示C与C++预处理器指令

string

如何显示字符串常量（如awk中的"Don't panic!"）

variable

如何显示变量、字段等内容

other

如何显示不是前述种类但不能以正常字体显示的东西（例如以C的typedef关键字定义的类型名称）

接下来，你指示字体外观，normal、bold、italic、underlined、emphasized、boxed、graphic、proportional或fixed的其中之一（这些可被缩写为单一字母）。然后你可以在字体外观后加上颜色。例如：

```
:color function bold yellow
```

每一种语言的注释、函数与关键字等的描述都存储在elvis.syn文件中。这个文件还包括许多种类的规范。例如，以下是针对awk的语法规范：

```
# Awk.  This is actually for Thompson Automation's AWK compiler, which is
# somewhat beefier than the standard AWK interpreter.
language tawk awk
extension .awk
keyword BEGIN BEGINFILE END ENDFILE INIT break continue do else for function
keyword global if in local next return while
comment #
function (
string "
regexp /
useregexp (,~
other allcaps
```

其格式很容易了解，在elvis的在线说明文档中也有详细解说。

elvis将字体与颜色关联到文件语法的不同部分，可以让打印出来的文件与屏幕上看到的相同（参阅第340页“显示模式”一节中对:lpr命令的讨论）。

在非位映射的显示设备上，如Linux的console，所有的字体都映射到console驱动程序所

使用的字体。此时若要分辨字体，如normal与italic，就很困难了。然而，在某些显示设备上，你仍然可以改变不同字体的颜色。如果你的GNU/Linux系统上有elvis，使用它随便编辑一个C源文件，即可看到各部分代码显示出不同颜色。彩色的语法效果很不错，但可惜印刷时不能呈现。

在elvis中，语法颜色是让窗口个别设置的属性。你可以在一个窗口中改变斜体字体的颜色，这并不会影响其他窗口中斜体字体的颜色。即使两个窗口显示相同的文件，也是如此。

语法着色可以让编辑器的工作更有趣且生动。但是你在选择颜色时可要小心！

有趣的功能

elvis有许多有趣的功能：

国际化支持

像nvi一样，elvis也有自己的方法，允许将消息翻译成不同的语言。elvis.msg 文件会依照elvis path搜索，并载入一个名为Elvis messages的缓冲区。

消息的格式是“简短消息：长消息”。在打印消息之前，elvis先寻找简短形式，如果有对应的长形式，则使用长消息，否则使用简短消息。

显示模式

这可能是最有趣的elvis功能。对某些种类的文件，elvis会在屏幕上对文件内容做格式化，得到近似“所见即所得”的惊人效果。elvis也可以用相同的格式化方法，在某些种类的打印机上打印缓冲区。关于显示模式，后面有一小节会单独介绍。

操作前与操作后的命令文件

elvis会载入4个文件（如果存在的话），用于自定义读、写文件前后的行为。这个特性也将另有小节介绍。

开放模式（open mode）

elvis是唯一真正实现了vi开放模式的同类品（可以将开放模式想象成只有单行窗口的vi。开放模式的“优点”在于可用在没有光标移动功能的终端上）。

安全性

:safer命令设置了security选项，以执行非主目录中的.exrc文件。设置security=safer 时，“某些命令不能执行，文件名称中的通配符扩展不能使用，某些选项会被锁定（包括security选项本身）”。这在elvis 的说明文档中有详细介绍，然而，不要盲目相信elvis会提供完全的安全性。

内置计算器

elvis扩展了ex命令语言，加入了内置的计算器（在说明文档中有时被称为表达式求值器〔expression evaluator〕）。它了解C的表达式语法，最常被用在:if、:calc与:eval命令中。要了解其细节，可以参阅在线帮助。范例位于elvis发行包的样本初始化文件中。

宏调试器

elvis有个vi宏（:map命令）调试器。在编写复杂的输入或命令映射时很有用处。

ex模式的宏

:alias命令用于定义ex宏。其意图是在csh中汇编alias命令。例如，:safely命令有别名为:safer，提供对elvis较早版本的向下兼容性。

较聪明的%命令

%命令已被扩展至可认别#if、#else、#endif指令，如果在syntax显示模式中使用的话。

内置的拼写检查

在syntax显示模式中，拼写检查足以应付tags文件中的程序符号或注释中的自然语言词汇。请参考:help set spell。

文本折叠

文本折叠可用于隐藏或显示文件的某些部分，在处理结构性文本时很有帮助。请参考:help :fold。

已选择行的高亮显示

Steve告诉我们："elvis可对已选择行增加高亮显示效果。请参考:help :region。例如，:load since加上:rcssince命令，将能使文件自上次存储入RCS后所有被改变了的行高亮显示。

显示模式

elvis有许多显示模式。依照文件种类的不同，elvis产生文件的格式化版本，有"所见即所得"的效果。表17-11概述了各种显示模式。

表17-11：elvis 显示模式

模式	显示样式
normal	不做格式化，照文件中文本的原样显示
syntax	像normal一样，但是打开语法着色

表17-11：elvis 显示模式（续）

模式	显示样式
hex	交互式的十六进制转储，令人回想起大型计算机上的十六进制转储。适合编辑二进制文件
html	简单的网页格式化。标签命令可用于点击链接或返回起点
man	简单的手册页格式化。类似nroff -man的输出
tex	TeX格式的简易子集

:normal命令可从前述显示模式切换回normal模式。使用:display *mode*可切换回原先的模式。^W d命令则是在窗口的显示模式间切换的捷径。

在所有可用的模式中，html与man的本质最符合“所见即所得”。在线说明文档中清楚地定义了elvis对这两种标记语言所理解的部分。

由于elvis的在线说明文档是用HTML编写的，其中有许多交叉引用的链接，故elvis使用了html模式来显示。

下例中，elvis正在编辑一份HTML格式的帮助文件。屏幕被分割成两部分，两个窗口显示同一个缓冲区。下面的窗口使用html显示模式，而上面的使用normal显示模式：

```
<html><head>
<title>elvis 2.0 Sessions</title>
</head><body>
<h1>10. SESSIONS, INITIALIZATION, AND RECOVERY</h1>

This section of the manual describes the life-cycle of an
edit session. We begin with the definition of an
<a href="#SESSION">edit session</a> and
what that means to elvis.
This is followed by sections discussing
<a href="#INIT">initialization</a>
and <a href="#RECOVER">recovery after a crash.</a>
______________________________________________________________

10.0 SESSIONS, INITIALIZATION, AND RECOVERY

        This section of the manual describes the life-cycle of an
        edit session. We begin with the definition of an edit
        session and what that means to elvis. This is
        followed by sections discussing initialization and
        recovery after a crash.

10.1 Sessions
```

man模式也很有趣，正常情况下，我们需要格式化手册页并打印出来，才能确定页面排版是否恰当。以下是引用自在线帮助文件的文本，看起来还不错：

> Troff source was never designed to be interactively edited, and although I did the best I could, attempting to edit in man mode is still a disorienting experience. I suggest you get in the habit of using normal when making change, and man monde to preview the effect of those changes. The ^W d commands switching between modes a pretty easy thing to do.

另外一件有趣的事情是html与man模式都可以与第337页“语法高亮显示”一节中描述的:color命令一起使用。这对man模式特别有帮助。例如，在 Linux console的默认情况下，只有黑体文本（.B）可以与正常的文本区分。但如果加上语法着色，则黑体与斜体（.I）文本都可以区分了。表17-12整理了显示模式的命令。

表17-12：elvis 的显示模式命令

命令	功能
di[splay][*mode* [*lang*]]	将显示模式改变为*mode*。使用*lang*来表示syntax模式
no[rmal]	与:display normal相同，但是比较容易输入

与各个窗口相关的是bufdisplay选项，它应该设置成某一种elvis支持的显示模式。标准的elvis.arf文件（参阅下一小节）会查看缓冲区文件名的扩展名，并尝试设置比normal更合适的模式。

最后，elvis也可以应用“所见即所得”的格式化方式，用于打印缓冲区的内容。:lpr命令格式化的范围为一行（或是整个缓冲区，这是默认值）。你可以打印到一个文件中或是进入命令管道。默认情况下elvis会打印到一个执行系统池打印命令的管道。

:lpr命令由数个选项所控制，如表17-13所述。

表17-13：elvis 的打印管理选项

选项	功能
lptype, lp	打印机的类型
lpconvert, lpcvt	如果设置的话，则将Latin-8的扩展ASCII 转换为PC-8的扩展ASCII
lpcrlf, lpc	打印机需要在每一行的最后加上CR/LF
lpout, lpo	打印至此文件名或命令
lpcolumns, lpcols	打印机的宽度

表17-13：elvis 的打印管理选项（续）

选项	功能
lpwrap, lpw	模拟行绕回（wrapping）
lplines, lprows	打印机一页的长度
lpformfeed, lpff	在最后一页后送出一个form feed
lpoptions, lpopt	控制许多打印机功能。只对PostScript打印机有用
lpcolor, lpcl	允许PostScript与MS Windows打印机进行彩色打印
lpcontrast, lpct	控制浓淡与对比，与lpcolor选项一起使用

大部分的选项都可以顾名思义。elvis支持的打印机类型请参考表17-14。

表17-14：lptype 选项的值

名称	打印机类型
ps	PostScript，一张纸一页
ps2	PostScript，一张纸两页
epson	大部分的点阵打印机不支持图形字符
pana	Panasonic的点阵打印机
ibm	可用于打印IBM图形字符的点阵打印机
hp	惠普打印机以及大部分非PostScript的激光打印机
cr	行打印机，用换行来进行重打
bs	用退格字符做重打。这种设置与传统的 Unix nroff 最接近
dumb	纯ASCII文本，没有字体的控制

如果你使用PostScript打印机，务必在lptype中使用ps或ps2。后者可以节省纸张，尤其是在打印草稿时特别有用。

操作前与操作后的控制文件

elvis让你可以在4个时间点控制文件的读写：读入文件的前后与写入文件的前后。系统分别在4个对应的时间执行4个不同的ex脚本。这些脚本是从elvispath选项所列出的目录中查找出的：

elvis.brf

这个文件在读入文件前（Before Reading File，.brf）执行。默认情况下会检查文件的扩展名并尝试猜测其是否为二进制文件。如果是的话，则会打开binary选项，

以防止elvis将新行符号（可能是实际的CR/LF对）在内部转换为换行符号。

elvis.arf

这个文件在读入文件后（After Redaing File，.arf）执行。默认情况下会检查文件的扩展名，以便打开语法高亮显示。

elvis.bwf

这个文件在写入文件前（Before Writing File，.bwf）执行，特别是在用缓冲区的内容完全替换掉原来的文件前。默认为复制原来的文件并存储到以.bak为扩展名的文件中。这必须设置backup选项。

elvis.awf

这个文件在写入文件后（After Writing File，.awf）执行。虽然在这里执行源码控制系统是个不错的主意，但并没有默认的文件。

使用命令文件来控制这些动作的效果非常强大。你可以轻易地修改elvis的动作，以符合你的需要。在其他编辑器中，这些特性大部分都固定在代码中。

除此之外，elvis用:autocmd支持Vim风格的自动命令。请参考在线帮助以取得详细信息。

elvis的未来

Steve Kerkendall告诉我们，他还实现了几个功能，但尚未发布：

- 给GDB（GNU 调试器）的界面，以便于软件开发
- 类似Vim的viminfo文件的持久性功能
- 在语法里嵌入其他语法的功能，例如在HTML里可嵌入JavaScript

资源与支持的操作系统

elvis的官方网站地址是*ftp://ftp.cs.pdx.edu/pub/elvis/README.html*。可从此下载 elvis 的发行包，或用ftp直接从*ftp://ftp.cs.pdx.edu/pub/elvis/elvis-2.2_0.tar.gz*处取得。

elvis的源码可以自由发布，它依据perl的Artistic License条款发布，授权条款请参见发行包中的doc/license.html。

elvis可在Unix、OS/2、MS-DOS及现在的MS Windows上运行。对Unix与Windows平台均提供图形用户界面。MS-DOS版本则有鼠标支持。

编译elvis的过程非常直接。通过ftp或浏览器取得发行包，将其解压缩并逐个打开文件之后（注2），运行configure程序，接下来运行make：

```
$ gzip -d < elvis-2.0.tgz | tar -xvpf -
...
$ cd elvis-2.0; ./configure
...
$ make
...
```

elvis应该会顺利配置并编译。最后使用make install来安装。

注意：默认配置使elvis自行安装在标准系统目录里，例如/usr/bin、/usr/share……如果希望把程序安装到/usr/local，请对configure脚本使用--prefix选项。

如果你需要报告elvis的错误或问题，可以联络Steve Kirkendall，他的电子邮件地址为*kirkenda@cs.pdx.edu*。

注2：可至elvis ftp站点取得untar.c程序，它能在非Unix系统上解开用gzip压缩的tar文件，是个跨平台又简单的程序。

第十八章

vile：类似Emacs的vi

vile代表示“类似 Emacs的vi”（vi Like Emacs）。它起源自3.9版的 MicroEMACS，并修改为“手感”类似vi。Tom Dickey与Paul Fox是它的维护者。自vile问世以来（1990 年开始），也曾有过其他贡献一己之力的维护人员，包括Kevin Buettner与Clark Morgan。

当前的版本是9.6，于2007年底发布。本章使用的屏幕截图来自9.5s版（预发布的 beta 版）。20世纪90年代晚期以前，vile的版本编号大约一年前进1号，从1999年开始，变成一年前进0.1号——某天它总会到达第10版。

本章用vile编写。

作者与历史

Paul Fox是这样介绍vile的早期历史：

> vile的设计目标总是与其他的vi同类品有点不一样。vile一直不打算成为一个 vi 同类品，即使很多人都认为它够像了。在1990年，我希望能够在多个窗口中编辑多个文件。当时我已经用了十年的vi，而在 Micro-EMACS的源代码溜过我的新闻阅读程序时，手边正好有太多的时间，这是 vile 诞生的原因。刚开始只是简单地更改现有的键映射，接着便全力解决“嘿！‘插入模式’在哪里？”之类的问题，因此就再多钻研一些、再多钻研一些，最后就在1991 或1992年发布了（此后，主版本编号即随着年份改变，7.3是1997年中的第三个版本）。
>
> 但是我的目标一直是保持“手感”（而不是视觉化的显示）。自私一点说，是保持我常用的大部分命令的手感。vile有惊人的ex模式，工作得非常好——只是看起来很奇怪，而许多超出当前解析器范围的命令都还欠缺。同理，vile也不能完全解析现有的.exrc文件，因为我并不认为有那么重要——它可以做简单的解析，但是比较复杂的就要花些工夫了。然而当你深入vile的内置命令 / 宏语言中时，很快就会忘掉.exrc了。

Tom Dickey在1992年12月时开始加入vile的开发，一开始只是提供补丁程序，后来加入了一些重要的特性与扩展功能，像行编号、名称自动补全、动态的缓冲区列表窗口等等。Tom表示："在我的设计目标中，整合各种功能比实现一大堆的功能要来得重要。"

1994年2月，Kevin Buettner开始加入vile的开发。他最初只提供了X11版本——xvile的错误修正，接着加入了一些功能改进，如滚动条。最后发展成支持Motif、OpenLook 与Athena widget set。因为Athena widget（令人惊讶地）不能"完全能以无错误的形式取得"，他便编写了一个使用原始Xt 工具集的版本。这个版本最后提供了比Athena版更优越的功能。Kevin也对vile的GNU Autoconf 开发作了最初的支持。

Win32 GUI平台称为winvile，从1997开始发展并持续加上扩展功能，包括OLE服务器与Visual Studio add-in。

在当前的vile版本中，perl界面与主要模式（稍后另有讨论）都已稳固。它们作为其他功能的基础，例如服务器（使用perl界面）以及根据主要模式进行的语法高亮显示。最近开发的焦点将放在本地支持的改良上。

重要的命令行参数

虽然vile并不预期会作为vi或ex来调用，但是它可以作为view调用，此时会将文件视为只读的。vile与其他同类品不一样，没有行编辑器模式。

以下是vile的重要命令行参数：

`-c` *command*, + *command*
: vile将执行指定的ex样式的命令。给定的-c选项没有数量限制。

`-h`
: 调用vile的帮助文件。

`-R`
: 以"只读模式"调用vile，此时不允许任何写入（当vile作为view被调用，或在启动文件中设置了readonly模式时也会如此）。

`-t` *tag*
: 从指定的*tag*处开始编辑。-T选项的作用相同，可以在X11选项解析将-t吃掉时使用。

`-v`
: 打开vile的"view"模式，在此模式下不会允许对任何缓冲区的改变。

-?

vile会显示出简单的使用说明，然后离开。

@cmdfile

vile会将指定的文件当成启动文件来执行，并跳过任何正常的启动文件（如.vilerc）或环境变量（如VILEINIT）。

有少量的常用选项自从vile实现了POSIX的-C（或+）选项后就废弃了：

-g *N*

vile会从第一个文件中指定的行开始编辑。也可以用+*N*来表示。

-s *pattern*

vile会在第一个文件中执行指定模式的初始搜索。也可以用+/*pattern*来表示。

在线帮助与其他说明文档

vile附上了一个（很大的）ASCII文本文件，即vile.hlp。:help命令（可以简写成:h）为此文件打开一个新窗口。接着你可以用标准的vi搜索技巧，来搜索某个主题的信息。因为这是一个普通ASCII文件，打印出来阅读也很简单。

除了帮助文件之外，vile有许多内置的命令，可以显示编辑器的功能与状态信息。其中一些最有用的命令包括：

:show-commands

创建一个新窗口，显示所有vile命令的完整列表，并附上简短的描述。这些信息都位于各自的缓冲区，因此可以当成其他的vile缓冲区一样使用。特别是，要将其写到文件中作为打印之用也很容易。

:apropos

显示所有名称中包含给定字符串的命令。这比起在帮助文件中随机搜索某个特定主题的信息要容易得多。

:describe-key

提示输入一个键或按键序列，接着显示这个命令的描述。例如，x键会实现delete-next-character函数。

:describe-function

提示函数名称，接着显示函数的描述。例如，delete-next-character函数会删除光标当前位置右边指定数目的字符。

:apropos、:describe-function与:describe-key命令都会产生描述性信息，加上所有的同义命令（因为一个函数可能为了方便起见有多个名称）、所有其他绑定的按键（因为许多按键序列可能会被绑定到同一个函数）以及此命令是一个“动作”还是“运算符”。下面这个:describe-function next-line的输出就是一个极好的例子：

```
"next-line"                    ^J      j       #-B
  or    "down-arrow"
  or    "down-line"
  or    "forward-line"
  (motion:  move down CNT lines )
```

它显示了所有的四个名称及其按键绑定（在vile中，#-B是与终端无关的向上箭头表示法——使用:show-key-names取得完整列表）。

VILE_STARTUP_PATH环境变量可设置成以冒号分隔的帮助文件搜索路径（注1）。VILE_HELP_FILE环境变量可用于覆盖帮助文件的名称（通常是vile.hlp）。

可搜索的在线帮助文件、内置命令与按键描述的结合，加上命令自动补全功能，使得帮助功能的使用非常简单。

初始化

vile与xvile会执行以下的初始化步骤：

1. （只适用于xvile）。如果有XVILE_MENU环境变量，则将其作为菜单描述文件的名称；否则，使用.vilemenu文件。这个文件的目的是设置X11界面的默认菜单（注2）。

接下来，不同版本的 vile、xvile、winvile 均执行相同的两阶段初始化。第一个阶段混合使用环境变量与文件：

2. 如果命令行中以@cmdfile指定了文件，即执行该文件，并跳过任何其他状态下会执行的初始化步骤。
3. 如果有VILEINIT环境变量，就执行其值；否则，查找初始化文件。
4. 如果有VIL_STARTUP_FILE环境变量，便将其当作启动文件的名称；如果没有，在Unix上会使用.vilerc，而其他系统上会使用vile.rc。

注1： Win32平台以分号分隔列表，OpenVMS则使用逗号。

注2： winvile的菜单不可配置，它们只提供Win32中支持的功能。

5. 寻找当前目录中的启动文件，然后在用户主目录中寻找。哪一个先被找到就使用哪一个。

第二个阶段使用初始化命令：

6. 载入命令行上指定的第一个文件到内存缓冲区。
7. 执行命令时加上-c选项，默认情况下应用命令至第一个文件。

与其他同类品一样，vile将共同的初始化动作放在.exrc文件中（例如针对Unix vi与其他同类品的选项与命令），并允许使用.vilerc来在vile专用的初始化之前或之后执行：source .exrc。

多窗口编辑

vile与其他同类品有些不同。它诞生时是一种MicroEMACS，后来才修改成与vi有相同"手感"的编辑器。

Emacs一直都能处理多个窗口与多个文件，因此vile是第一个提供多窗口与多编辑缓冲区的vi类似程序。

就像在elvis与Vim中一样，:split命令（注3）创建一个新窗口，然后你可以使用ex命令:e *filename*在新窗口中编辑新文件。但接下来就不一样了，特别是在vi命令模式中切换窗口的按键非常的不一样。

图18-1显示了输入vile ch12.xml（注4），然后输入:split，再输入:e !zcat chapter.xml.gz后的屏幕分割效果。

所有的窗口都共享执行ex命令的最底行，与 Vim 一样。每一个窗口有自己的状态行，当前窗口的状态行中填满等于号。如果处于插入模式，状态行的第二栏会显示I。如果文件已经改变，但是还没有写入，则会再加上[modified]。

vile的命令与按键序列绑定，这一点与Emacs类似。表18-1显示了命令与对应的按键序列。有时候，两组按键序列会做相同的操作，例如delete-other-windows命令。

注3：它是经vile加工后的成品，以便缩短命令。实际的命令是split-current-window。

注4：有些读者可能发现本章并非第十二章。这是因为在编写本书第7版的过程中，我们调整了一些章节顺序。

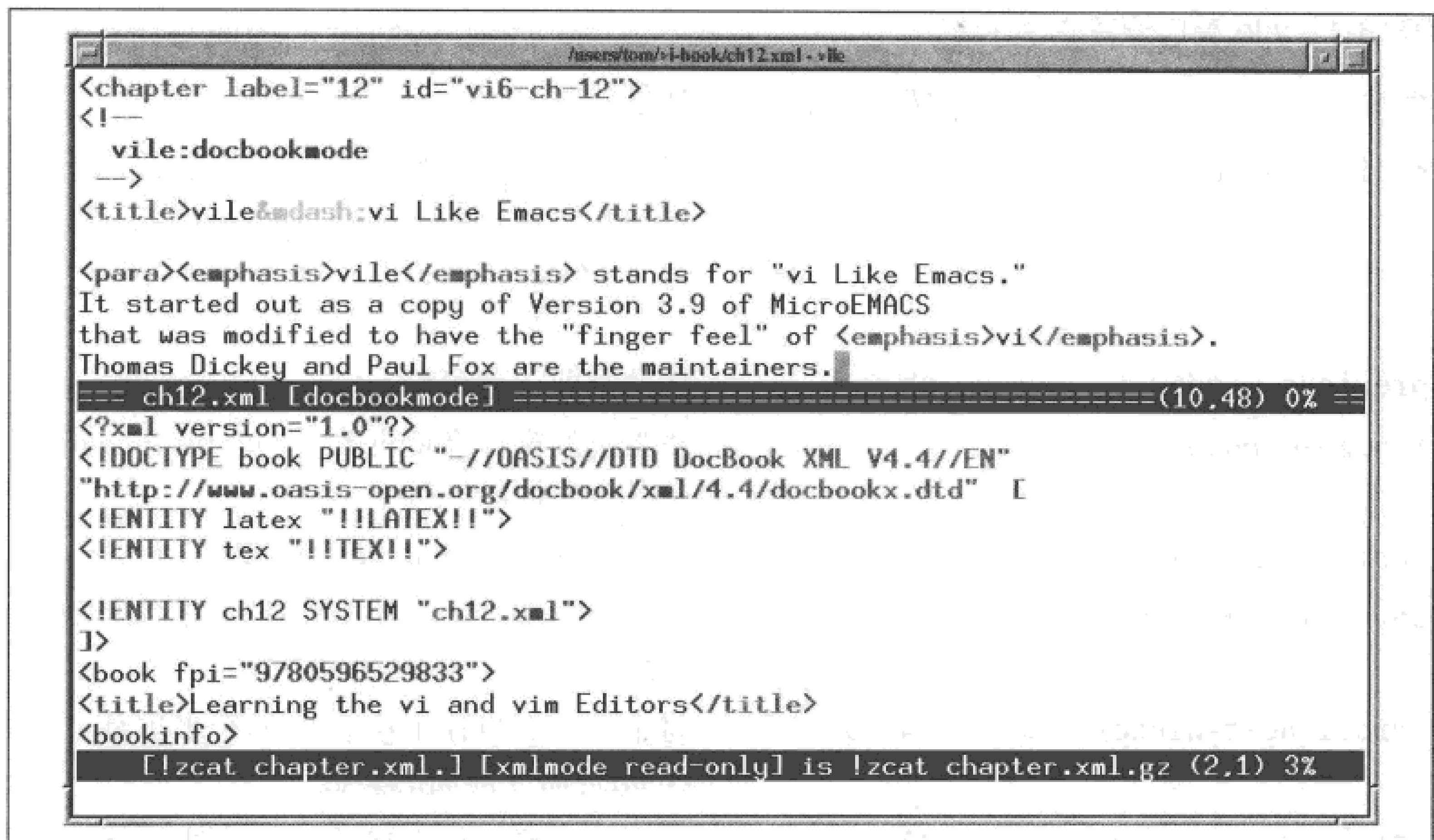


图18-1：在vile中编辑此章

表18-1：vile 的窗口管理命令

命令	按键序列	功能
delete-other-windows	^O，^X 1	除了当前的窗口之外消除所有的窗口
delete-window	^K，^X 0	终结当前窗口，除非它是最后一个
edit-file，E，e	^X e	把指定文件（或光标下的文件，使用^X e）或现有的缓冲区放入窗口中
find-file	^X e	同edit-file
grow-window	V	将当前窗口的尺寸增加*count*行
move-next-window-down	^A ^E	将下一个窗口往下（或把缓冲区往上）移动*count*行
move-next-window-up	^A ^Y	将下一个窗口往上（或把缓冲区往下）移动*count*行
move-window-left	^X ^L	将窗口往左滚动*count*栏。如果没有指定，则滚动半个屏幕
move-window-right	^X ^R	将窗口往右滚动*count*栏。如果没有指定，则滚动半个屏幕
next-window	^X o	移到下一个窗口

表18-1：vile 窗口管理命令（续）

命令	按键序列	功能
position-window	z where	根据*where*指定的光标位置，重新决定窗口位置，*where*值可以是：中心（.、M、m）、顶端（ENTER、H、t）或底端（-、L、b）
previous-window	^X O	移动到前一个窗口
resize-window		将当前的窗口高度更改为*count*行。*count*是命令前面的参数
restore-window		回到save-window所保存的窗口
save-window		标记一个窗口，以便使用restore-window回到该窗口
scroll-next-window-down	^A ^D	将下一个窗口往下移动*count*个半屏幕。*count*是命令前面的参数
scroll-next-window-up	^A ^U	将下一个窗口往上移动*count*个半屏幕*count*是命令前面的参数
shrink-window	v	将当前窗口的尺寸减少*count*行。*count* 是命令前面的参数
split-current-window	^X 2	将当前的窗口分成两半。*count*值为1或2，用于选择当前的窗口。*count*是命令前面的参数
view-file		将指定的文件或现有的缓冲区放入窗口中并标记成“只读的”
historical-buffer	_	显示前9个缓冲区的列表。按数字键可移到对应的缓冲区，而按_会移到最近编辑的文件。按tab键（或退格键加tab键）可轮流显示列表项，若遇到缓冲区名称较长的列表，这样比较容易在列表项间移动
toggle-buffer-list	*	弹出一个窗口，显示所有的vile缓冲区

图形用户界面

本节中的屏幕截图及其解释是由Kevin Buettner、Tom Dickey、Paul Fox所提供的，非常感谢他们。

vile有许多种X11界面，各自使用以Xt函数库为基础的不同工具集（toolkit）。有一个

称为“不用工具集”（No Toolkit）的版本没有使用工具集，但是它拥有自定义的滚动条与布告栏式的widget以管理geometry。还有使用 Motif、Athena 或 OpenLook toolkit的版本（注5）。Motif与Athena版本的支持最好，而且有功能菜单的支持。

有“一个”Win32 GUI ——并有支持 OLE 与 Unicode 的变异版本。它们表面上看起来一样。

幸好，每一种版本的基本界面都相同。有一个单一的最上层窗口，可以分成两个或多个窗格（pane）。窗格也许用来显示同一个缓冲区的多个窗口，或是多个缓冲区，或是两者皆有。在vile的说法中，这些窗格称为“窗口”（windows），但是为了避免混淆，在下面的讨论之中，我们会继续将其称为“窗格”。

构建xvile

虽然有xvile的二进制包，但你或许会在没有包支持的平台上编译它。

要构建xvile，你必须选择使用的工具集版本。这是在用configure命令调整vile的配置时完成的（注6）。相关的选项如下：

`--with-screen=`*`value`*
: 指定终端驱动程序。默认值是tcap，它是termcap/terminfo的驱动程序。其他值包括ncurses、ncursesw、X11、OpenLook、Motif、Athena、Xaw、Xaw3d、nextaw与ansi。

`--with-x`
: 使用X Window System 。这是所谓的“不用工具集”的版本。

`--with-Xaw-scrollbars`
: 使用Xaw滚动条，而不是vile的自定义滚动条。

`--with-drag-extension`
: 以Xaw使用拖动／滚动扩展功能。

基本的xvile外观与功能

这些图显示了xvile的Motif界面，与Athena界面很像。

注5：Sun Microsystems在发布Solaris 9（2002年）前，撤销了对OpenLook的支持。

注6：configure脚本应该能在任何Unix（或类似）平台上运行。要在OpenVMS上构建xvile 时，请使用vmsbuild.com脚本。构建此程序的指示说明，在脚本最上面的注释中。

图18-2显示了三个窗格：

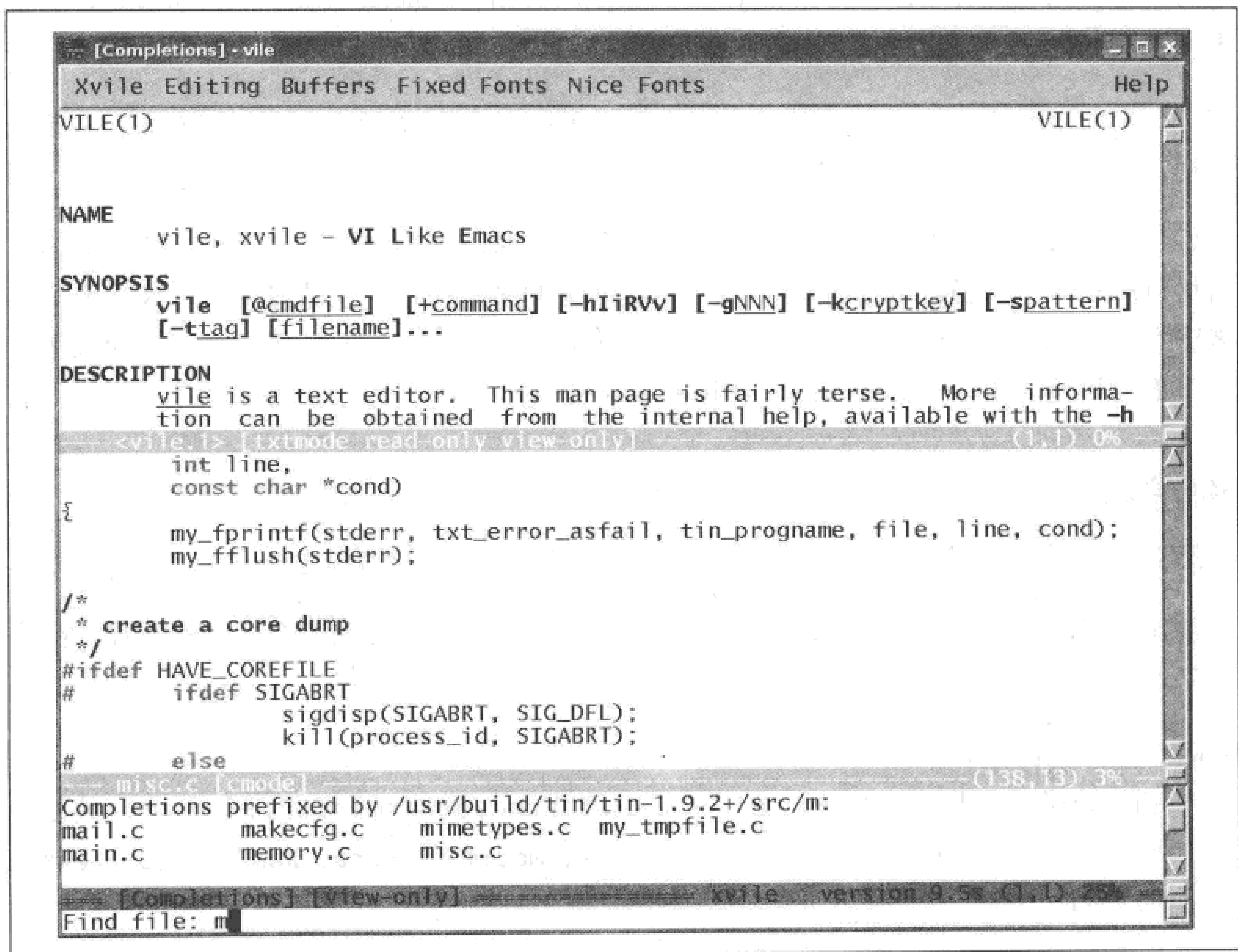


图18-2：xvile的GUI窗口

1. vile手册页，显示了下划线与黑体的使用。
2. 一个显示了tin的缓冲区misc.c（加上了语法高亮显示——印刷版为灰度—— 标示出预处理器语句、注释、关键字）。
3. 一个三行的窗格，当前是活跃的（由较暗的状态行得知），命名为[Completions]，作为文件名补全之用。该窗格与迷你缓冲区（冒号命令行）相配合：第一行会显示Completions prefixed by /usr/build/in/tin-1.9.2+/src/m:，而迷你缓冲区则显示Find file: m。窗格的其他部分包含了实际匹配的文件名。[Completions]的第一行及其内容会在用户补全文件名（按下TAB告诉vile显示缩小范围后的选择）之后改变。

图18-3也显示了三个分区：

1. [Help]分区，当然显示了一个编辑器最重要的功能（如何离开而不修改文件）。

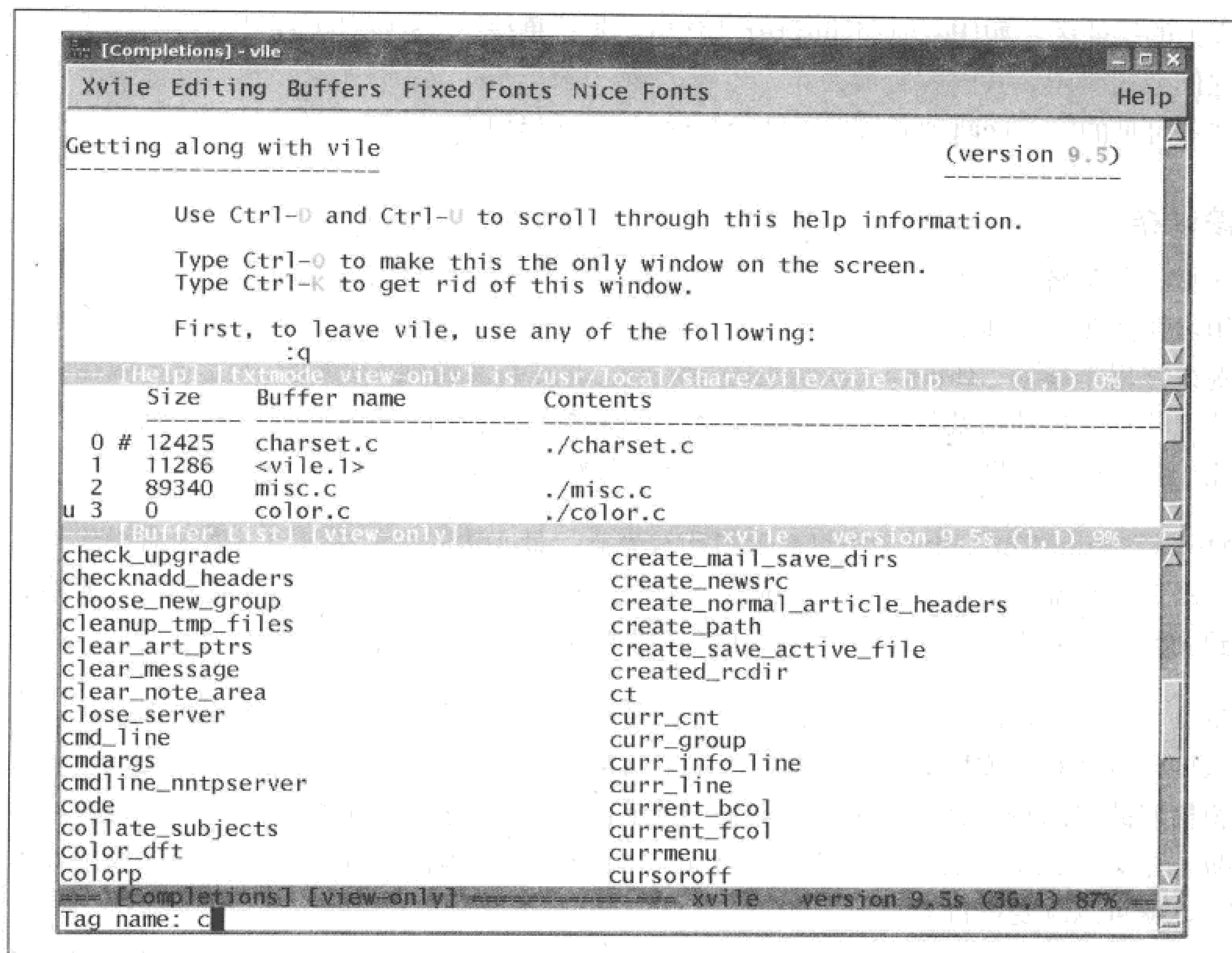


图18-3：vile中的缓冲区与自动补全

2. [Buffer List]，指示charset.c是#（上一个）缓冲区。%（当前的）缓冲区没有显示在列表中，因为只有“可见的”缓冲区会显示在这一份[Buffer List]中。对 * 命令提供一个参数，可显示不可见的缓冲区。缓冲区0与2是charset.c与misc.c，它们均已经被载入了，因此其大小（12425与89630）会显示在[Buffer List]中。Buffer 1（<vile.1>）中有已格式化的手册页，该页面由宏产生，且未对应至文件（注7）。Buffer 3（color.c）还没有载入，因此在最前面显示u，其大小则为零。

3. [Completions]缓冲区是活跃的。此处是对部分匹配*co*的标签所做的自动补全，而因为缓冲区已经往下滚动，“Completions prefixed”的消息并没有显示，这是另一个按下TAB的副作用：vile通过滚动的动作显示所有选择，即使当窗口很小时也一样（注8）。

注7：　在名称<vile.1>中的角括号是避免命令冲突的惯例，因为两个缓冲区不能有相同的名称。

注8：　[Completions]缓冲区会自动调整大小，不占用超出需要的行空间。如果可用空间不够，vile则会向相邻窗格借用3/4的空间。

产生的缓冲区，如[Help]和[Buffer List]，是“草稿”（scratch）缓冲区。当压栈时，它们会被关闭，其内容会被丢弃。还有另外一些缓冲区，举例来说，它们包含脚本，是“不可见的”。这两种通常都不显示在[Buffer List]中。

滚动条

在每一个窗格的右侧均有滚动条，习惯上用于缓冲区里的移动。然而，请注意，它的使用习惯将随着工具集的不同而不同。在Athena与“不用工具集”的版本中，鼠标的中间键或许用于拖动“thumb”或可见的指示器。左键与右键则（分别）在缓冲区中往下或往上移动。移动的量依鼠标位于滚动条上的位置而定。如果位置靠近顶端，则滚动量可能少到一行；如果位置靠近底端，则可能滚动整个窗格。

Motif与OpenLook的滚动条可能显得比较熟悉。最左边的鼠标按钮可以做所有操作。在滚动条的小箭头上单击，会往上或往下移动一行。滚动条的指示器可以拖动，以便移动到各处；若要滚动整个窗格，则可在指示器的上方或下方单击。

每一种版本中，滚动条上方或下方（也就是在两个滚动条中间）有一个小把手，可以用来调整两个相邻窗格的大小。在“不用工具集”的xvile版本中，这个把手与两个相邻窗格间的状态行融合了。其他版本里，调整窗格大小的把手比较容易分辨。但是所有的版本中，当鼠标光标位于这个把手上时，会变成粗的纵向双箭头。在把手上单击并拖动，就可以改变窗口大小。

按住Ctrl键再用鼠标左键在滚动条上单击，便可以将窗格一分为二。接着你会看到这个缓冲区的两个视图。如果需要的话，使用其他的vile命令可以将其中一个窗格中的内容换成别的缓冲区。按住Ctrl键并用鼠标中键单击，便可以删除某个窗格。有时在创建许多窗格之后，你会想把所有窗口空间都给一个窗格使用，此时只要按住Ctrl键，再在某个窗格中单击鼠标右键，就会删除其他窗格，让整个xvile窗口中只包含鼠标所在的窗格。表18-2总结了本段所解释的动作。

表18-2：vile 的分区管理命令

命令	功能
Ctrl加鼠标左键	在滚动条上单击，分割窗格
Ctrl加鼠标中键	删除窗格
Ctrl加鼠标右键	让被单击的窗格成为唯一窗格

设置光标的位置与鼠标的动作

在窗格的文本区域中，可以按下鼠标左键来设置光标。这不只是设置光标位置，还会设置所编辑的窗格。如果只想设置窗格，但是保留原来的光标位置，则改为点击欲编辑文本区域下的状态行。

`vile`将鼠标的点击视同移动命令，例如要删除5行，你可以输入`d4j`，以删除当前这一行与下面的4行，也可以用鼠标完成同样的工作。将光标移到要开始删除的地方，再按 `d`。之后，在想要删除的终点单击鼠标。单击鼠标是真实的动作，可以与其他动作一起使用。

选择区域

选择区域，可由按住鼠标左键再拖动鼠标来完成，这称为PRIMARY选择。放开鼠标左键会使选择的文本被拉动并可作为粘贴（如果需要的话）之用。你可以在拖动鼠标时按住控制键，强迫选择区域维持矩形。如果拖动的动作超出当前的窗口，（可能的话）文本将自动往适当方向滚动，以配合超过窗口的选择范围。滚动的速度会随着时间而增加，以便在选择大范围文本时加快速度。

用鼠标双击或三击，即可选择个别单词或行。

按下鼠标右键可以扩展选择的范围。与左键的用法相同，按住鼠标右键拖动就可以调整或滚动选择的范围。在任何属于相同缓冲区的窗口中，只要其中之一开始了选择动作，就可以在其他窗口中做扩展。换句话说，如果在同一个缓冲区中有两个窗口（位于两个窗格中），其中之一包含了缓冲区的开头，另一个则包含结尾，有可能在显示缓冲区开头的窗格中于开头处单击鼠标左键，接着在显示缓冲区结尾的窗格中于其结尾处单击鼠标右键，就可以选择整个缓冲区。另外，选择的区域也可以扩展成矩形，只要在使用鼠标右键时按住Ctrl键即可。

鼠标的中键用于粘贴时选择范围。默认情况下是粘贴到上一个文本光标的位置。如果在按下鼠标中键时按住Shift键，则会粘贴到鼠标所在的地方。

在状态行上双击，可以清除选择范围（xvile拥有的选择范围）。

剪贴板

通过PRIMARY选择，可在许多X应用程序之间交换数据。这种选择方式在前面已经讨论过了。

其他应用程序会使用CLIPBOARD（剪贴板）选择在应用程序间交换数据。在许多Sun键盘上，可以按下COPY键将选择的文本移到剪贴板，按下PASTE键则可粘贴。如果你发

现xvile中选择的文本不能粘贴到其他的应用程序，或遇到相反情况，可能是因为这些应用程序使用了CLIPBOARD，而不是PRIMARY（在非常古老的应用程序中，则采取利用环状剪切缓冲区的机制）。

xvile提供了两个管理剪贴板的命令：copy-to-clipboard与paste-from-clipboard。在执行copy-to-clipboard时，当前选择的内容会复制到特殊的剪贴板删除寄存器（在寄存器列表中以;表示）。当应用程序要求使用剪贴板时，xvile即提供此删除寄存器中的内容。paste-from-clipboard命令会向当前CLIPBOARD的拥有者要求剪贴板中的内容。

Sun系统的用户可能有意把以下的按键绑定加入.vilerc中，以便使用键盘上的COPY与PASTE键：

```
bind-key copy-to-clipboard #-^
bind-key paste-from-clipboard #-*
```

按键绑定稍后将详细描述。

资源

xvile有许多可以用来控制外观与行为的资源。如果你想正确显示斜体字，字体的选择就非常重要。vile的说明文档中有完整的资源列表以及.Xdefault项目的样本集。

增加菜单

Motif与Athena版本都支持菜单。菜单项可以由用户定义，读入位置则由当前目录或主目录中的.vilemenu文件决定。

xvile 允许三种类型的菜单项：

- 内置项，也就是菜单系统专用的项目，如重新读入.vilerc文件或打开一个新的xvile副本。
- 内置命令的直接调用（例如显示[Buffer List]）
- 任意命令字符串的调用（例如运行交互式宏，像搜索命令）

后两种的区别，是因为作者首先让vile能够在执行前先检查命令的合法性。

构建winvile

winvile每次发布的二进制文件均可取得，但你或许打算编译其中某个过渡的补丁版本。源文件里提供了Microst（makefile.wnt）与Borland（makefile.tbc）编译器所用的

makefile。前者具有较多功能，构建时可用选项加上OLE、perl以及内置语法高亮显示。Win32 GUI也能在编译器环境里构建。

winvile的基本外观与功能

图18-4与18-5显示了winvile的Win32图形用户界面。从外表上看，它与“不用工具集”的X11界面很像，也有滚动条。表面之下（要接触到很容易），则比Motif界面复杂许多。

图18-4为 winvile编辑Unicode数据时的情况：

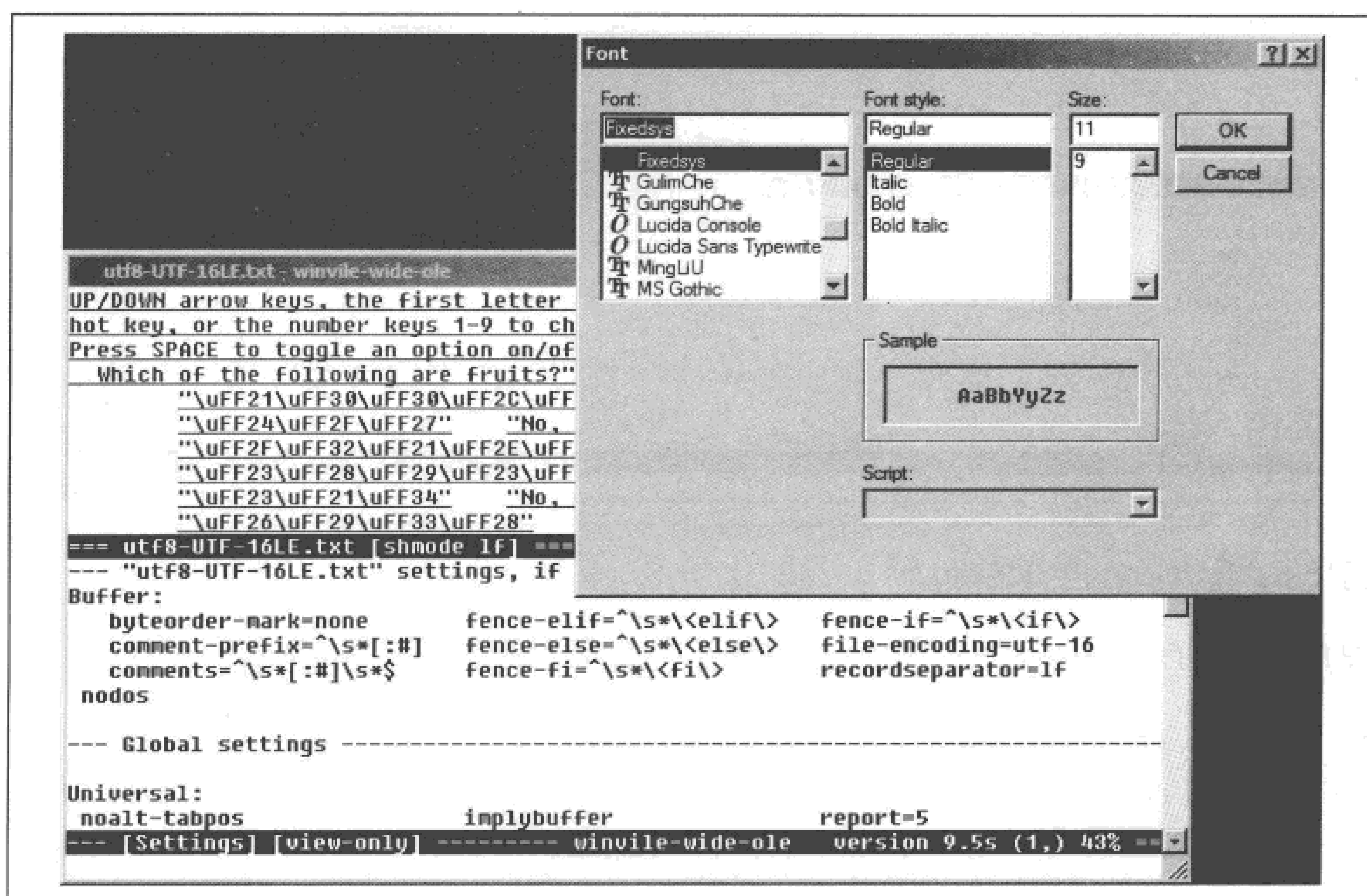


图18-4：具非Unicode字体的winvile

- 字体对话框最初设置为定宽系统字体。与xvile相同，字体能在启动winvile时设置，或通过脚本设置，也能通过OLE server设置。最后，如本图所示，亦可使用Win32上的常见控制方式。
- 数据为Unicode UTF-16，没有byte order mark。内容都被加上下划线，因为在高亮显示的配色方式中规定，使用下划线与亮蓝色为加上引号的字符串着色。
- 默认系统字体不能显示文件中的字符。winvile看到字体小，遂以十六进制的形式呈现Unicode数据。

图18-5显示了选择更适当字体后的效果。如果再次选择系统字体，winvile又会显示为十六进制形式。若首选以十六进制值显示宽字符（wide character），vile也有可供设置的选项。

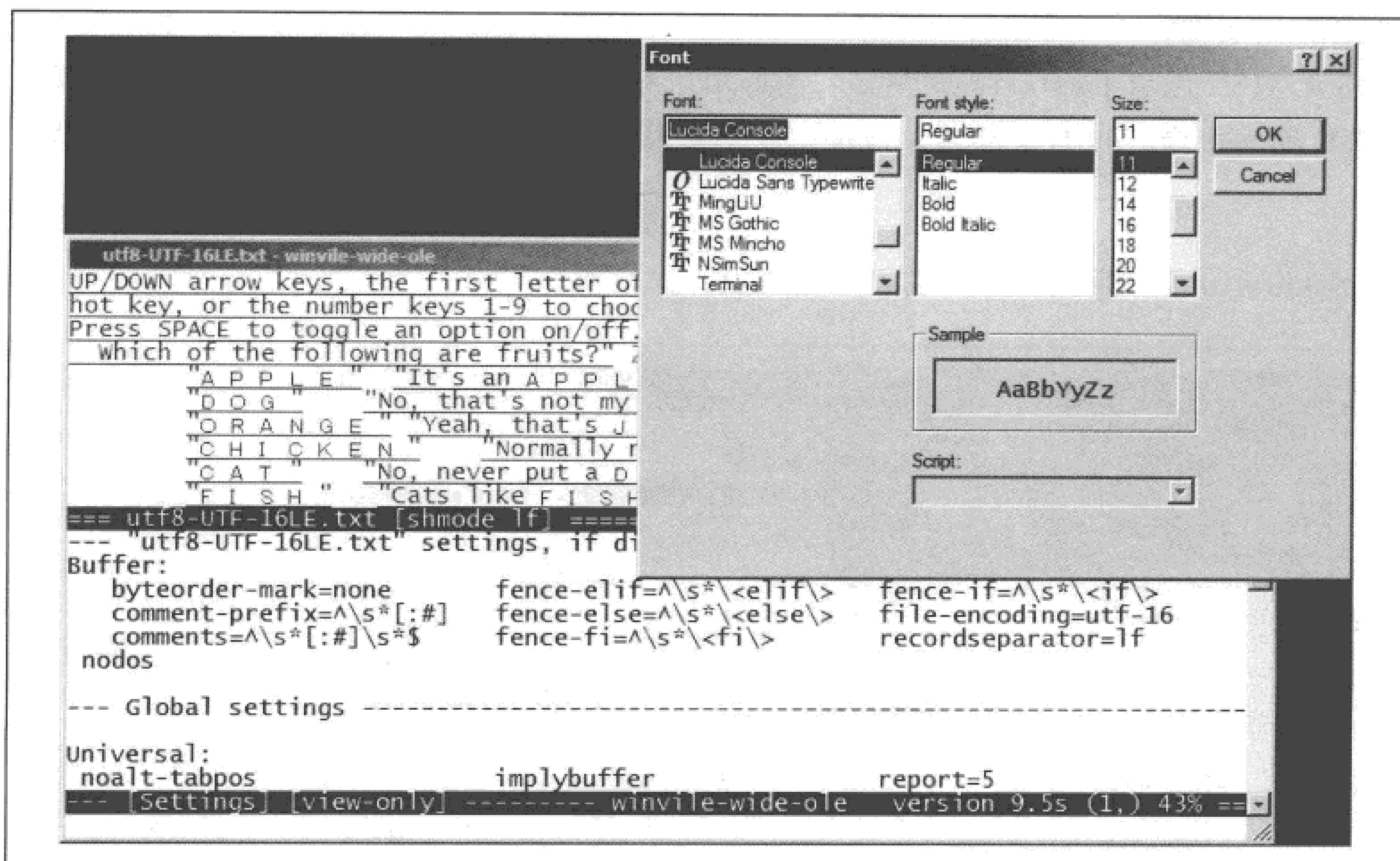


图18-5：具Unicode字体的winvile

图18-6显示了一些winvile菜单功能，包括：

- winvile扩展系统菜单，在窗口标题列上按下鼠标右键即可访问。在右键弹出的菜单里也有相同的选择，减少了上移到标题栏的必要性。请勾选菜单最下方的“Menu”项，这样按下鼠标右键即可弹出菜单。
- 菜单中提供打开、保存、打印、字体操作等常在图形用户界面应用程序里看到的操作项。我们也可以通过“CD”项设置winvile的当前工作目录。
- winvile也允许我们浏览Windows中的*Favorites*文件夹。
- 从用户可配置的“最近”文件（或文件夹）数量中选出的最近文件（与最近文件夹）条目。winvile于用户的注册表数据中保存名称，让每个运行中的winvile实例都能使用。

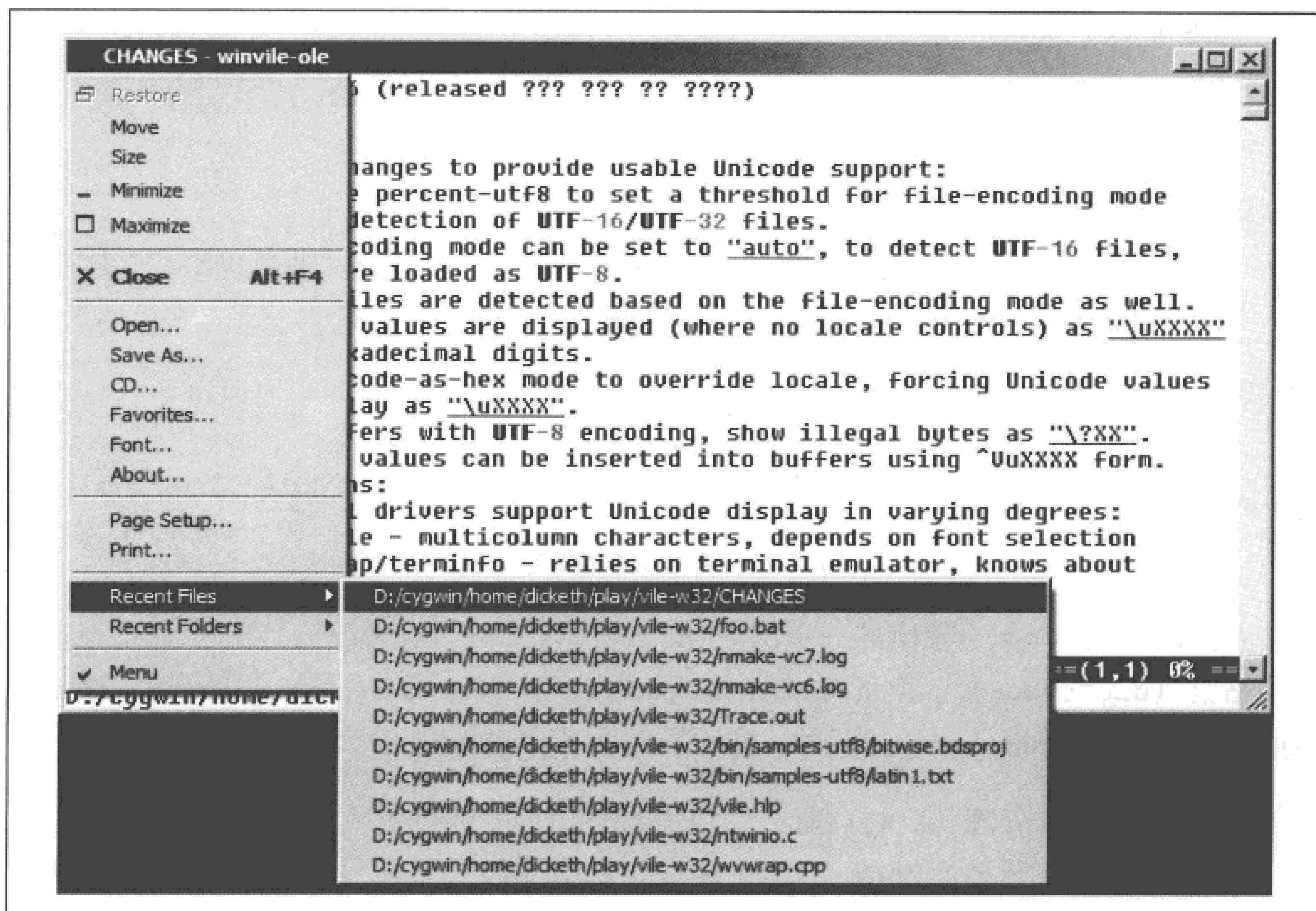


图18-6：winvile的“Recent Files”菜单

扩展正则表达式

扩展正则表达式稍早于在第八章的第131页的“扩展正则表达式”一节中介绍。vile提供了本质上与nvi的extended选项相同的功能。包括POSIX的用于字符类的方括号表达式，[[:alnum::]]，以及一些扩展功能（额外类与缩写），还有区间表达式，例如{,10}。但语法与nvi稍有不同，vile额外依赖反斜线转义字符：

\|

表示交替（alternation）：house\|home。

\+

前面的正则表达式需匹配一次或多次。

\?

前面的正则表达式需匹配零次或一次。

\(...\)

为*、\+、\?提供分组功能以及可以在替代命令中的替换部分（\1，\2……）中取

用匹配的子文本。

\s \S

分别可比较空白与非空白字符。

\w \W

分别匹配“单词组成”字符（字母数字与下划线字符）以及“非单词组成”字符。例如，\w\+ 可匹配 C/C++ 的标识符与关键字（注9）。

\d \D

分别匹配数字与非数字。

\p \P

分别匹配可打印字符与不可打印字符。空白视为可打印字符。

vile允许在替代命令的替换部分中出现\b、\f、\r、\t、\n等转义序列。它们分别表示退格符（backspace）、换页符（form feed）、换行符（carriage return）、制表符（tab）与新行符（newline）。另外，在vile的文件中也提到：

> 请注意vile在处理\u\L\1\E时，它模仿了perl而非vi的方式。同样是s/\(abc\)/\u\L\1\E/，vi将替换为*abc*，而vile与perl则替换为*Abc*。这对以大写字母开头的单词应该会比较有用。

改进的编辑功能

这一节描述了vile简化及增强的纯文本编辑工作的功能。

命令行历史记录与自动补全

vile会将你的ex命令存储在名为[History]的缓冲区中。这项功能由history选项所控制，默认值为true。如果将其关闭，则会禁用历史记录功能，并移除[History]缓冲区。show-history命令可分割屏幕，并在新窗口中显示[History]缓冲区。

冒号命令行实际上是个迷你缓冲区。可用于返回[History]缓冲区中的行并做编辑。

你可以用↑与↓在历史记录中往上或往下滚动，而←与→可在一行中移动。当前的删除字符（通常是BACKSPACE）可用于删除历史记录行中的字符。你输入的任何字符将被插入光标的当前位置。

输入mini-edit字符（默认是^G），可将迷你缓冲区切换到vi模式。此时vile会用mini

注9：追根究底的话，它也能匹配出以数字开头的标识符，通常这不是什么问题。

hilite选项中所指定的机制令迷你缓冲区高亮显示。其默认值是reverse（高亮）。在vi模式中，你可以使用vi风格的命令来移动位置。你也可以使用其他适于一行内容编辑的vile命令，例如i、I、a、A等。vile根据一份命令表决定可接受的命令，命令表也允许在迷你缓冲区里使用按键绑定（key binding）。

一个有趣的特性是，vile会使用历史记录列出你输入的命令所对应的既有数据。例如，在输入:set加上一个空格后，vile会提示Global value:。此时，你可以用↑显示已经设置过的全局变量并做改变。

ex命令行提供了各种补全的方法。当你输入命令名称时，可以在任何时候按下TAB键。vile 会尽可能填完剩下的命令名称。如果再按一次TAB，vile会创建一个新窗口，显示所有可能的补全方法。

自动补全可以用在内置与用户自定义的vile命令、标签、文件名、模式（本章后面会提到）、变量、枚举值（如颜色名称）与终端字符（如退格、中止等字符设置，由stty设置中取得）。

顺便说一下，这产生了一个有趣的现象。在vi风格的编辑器中，命令的名称可能很长，但是倾向于具有不同的开头字符，以便使用缩写。而在Emacs风格的编辑器中，命令名称开头的几个字符通常不会相异，但是命令的自动补全功能仍然可以省下大部分的打字时间。

标签栈

标签堆栈已在第134页的“标签栈”一节略做说明。vile可以使用标签堆栈，并且非常直接、简单。但它与其他同类品有些不同，最明显处在于标签搜索与弹出标签栈的vi模式命令。表18-3列出了vile的标签命令。

表18-3：vile的标签命令

命令	功能
next-tag	通过tags文件持续搜索更多匹配的内容
pop[!]	弹出栈中的一个光标位置，将光标恢复到前一个位置
show-tagstack	创建一个新窗口来显示标签栈。显示的内容随着标签被压入或弹出栈而变动
ta[g][!] [*tagstring*]	编辑包含tags文件定义的*tagstring*的文件。如果当前缓冲区已修改过但未保存，!会强迫vile切换到新文件

表18-4描述了vi模式命令。

表18-4：vile命令模式的标签命令

命令	功能
^]	在tags文件中查找光标所在位置标识符的位置，并移到找到的位置。当前位置会自动被压入标签栈
^T ^X ^]	回到标签栈中的前一个位置，也就是弹出一个元素
^A ^]	与:next-tag命令相同

像其他的编辑器一样，选项可以控制vile如何管理与标签相关的命令，如表18-5所示。

表18-5：vile的标签管理选项

选项	功能
pin-tagstack	让标签搜索与弹出时不改变当前窗口，即采用“钉住”（pinning）内容的方式。默认选项值为false
tagignorecase	让标签的搜索忽略大小写。默认值是false
taglength 符	控制查找的标签中的有效字符数量。默认值为零，表示所有字都是有效字符
tagrelative	若是在其他目录中使用tags文件，则tags文件中的文件名会被当成相对于tags文件所在的目录
tags	可以设置成用空白分隔的tags文件列表，以供查找标签。vile会将所有的tags文件载入独立的缓冲区，默认为隐藏的，但可于需要时编辑。你可以在tags中加入环境变量与shell通配符
tagword	用光标所在位置的完整单词作标签查找之用，而不只是从当前光标位置开始的一部分单词。默认为禁用这个选项，可让vile与vi兼容

不限次数的撤销

在这方面，vile原则上与其他的编辑器相似，但实践中不同。例如在elvis与Vim中，可以设置撤销的次数限制， nvi以.命令做下一次合适的撤销或重做动作。另外有单独的vi模式命令来实现连续的撤销与重做。

而vile使用undolimit选项来控制存储的改变次数。默认值为10，表示可以撤销最近10次的改变。如果设置成0，则成为真正的“不限次数的撤销”，但是会消耗掉很多的内存。

要启动撤销功能，第一次先使用u或^X命令，后续每执行一次.命令均会再做一次撤销。vile有一点类似vi：连续发出两次u命令，只会切换改变的状态；然而，每执行一次^X u命令，则会做一次撤销。

^X r命令会做一次重做。在第一次使用^X r后，后续每执行一次 . 命令均会再做一次重做。你可以在^X u与^X r命令前加上计数值，让vile执行指定数量的撤销或重做。

任意长度的行与二进制数据

用vile编辑的文件里，可包含任意长度以及任意数量的行。

vile会自动处理二进制数据，不需使用特别的命令行或选项。要输入8位的文本，可以在^V之后加上x与两个十六进制数字，或是o与三个八进制数字，或是三个十进制数字。

也可以输入16位的Unicode值，即在^V后加上u与最多4个十六进制数字。如果当前缓冲区的file-encoding选项属于Unicode之一（utf-8、utf-16或utf-32），vile将直接存储缓冲区为UTF-8，根据终端或显示设备的能力而呈现。

讲到 Unicode，引出了关于本地化的问题。

本地支持

多年以来，vile只有非常初步的本地支持，部分原因在于许多平台上的本地支持也很不成熟（除了Unix生产商）。它有自己的字符类型表（也就是控制字符、数字字符、可打印字符、标点符号，还有应用程序专用的文件名、通配符、shell字符），允许我们指定可打印的非ASCII字符。

时光流逝，vile也顺应用户的需求而发展。以下简短总结改变的事项，我们不是根据其开发出的时间而是依逻辑排列：

- 不是有一份固定的字符类型表，vile导入主机的字符类型表，再提供通过脚本修改数据的命令（注10）。
- vile的正则表达式支持POSIX字符类以及对应到vile自己的字符类型的类。
- vile支持从屏幕上抽取token，例如tags专用类、脚本专用类等等。这些token曾是特殊解析逻辑（parsing logic）下的字符类型测试的混合。现在，它们只是纯粹的正则表达式，不需要解析逻辑。

注10：即使在Unix系统上，这项功能也很有用，这些系统不见得总会传入正确的字符表。

- 编辑包含8位数据的文件时——例如以ISO-8859-7（Greek）编码的数据，若主机的本地编码使用UTF-8将是项挑战。当vile启动时，它检查主机本地设置的结尾是否为UTF-8（或类似 UTF-8），例如el_GR.UTF-8。如果与UTF-8有关，则支持以相应的8位本地编码做编辑，例如el_GR。
- 相似地，在支持UTF-8的主机环境中编辑文件，有编码为UTF-8的文件。在最新发布的版本中，我们可以要求vile以各种Unicode编码写入文件，并以相同编码读入。8位编辑模型也被带着向前走，为标记为8位的缓冲区翻译为8位编码，并可直接编辑（也就是不用翻译）Unicode缓冲区。

以上就是所有扩展，而每个阶段也都维持了原有功能。

vile还有一些尚未着手的本地化方面，例如消息格式化以及文本核对顺序。

文件格式

当vile读入文件，它对读入的内容先做数次猜测，目的是在我们面前呈现有用的数据：

- 检查文件的权限是否允许用户写入。
- 检查行尾（line ending），可能有CR、LF、CR/LF等数种可能。
- 检查Unicode字节的顺序标记。
- 检查Unicode的多字节编码。

根据上述检查，vile或许可以设置新读入缓冲区的特性（称为“模式”，mode），并应用到该缓冲区。此外，vile可能在读入时翻译数据：

- 移除每一行的行尾（line ending），记忆相关的recordseparator模式。
- 如果文件缺少最后一行的行尾，vile即设置nonewline选项。
- 把UFT-16与UFT-32数据翻译成UFT-8，记忆相关的file-encoding选项。

当我们命令vile把缓冲区写入文件时，它即使用上述本地选项设置以重新构建文件。

增量搜索

如第139页“增量搜索”小节所提到的，在vile中可以使用^X S与^X R命令执行增量搜索，不需要特别设置选项。

输入时，光标会在文件中移动，总是会位于匹配文本的第一个字符上。^X S用于往前的增量搜索，而^X R用于往回的增量搜索。

你可能希望将一些命令加到.vilerc中（说明请见第372页的“vile编辑模型”一节），

以便让熟悉的/与?命令也可以执行高亮显示：

```
bind-key incremental-search /
bind-key reverse-incremental-search ?
```

另一个引人注目的焦点功能是“视觉匹配”，令所有匹配的事物高亮显示。例如，在.vilerc文件中加入：

```
set visual-matches reverse
```

这个命令指示vile在视觉匹配时使用高亮。因为高亮显示有时会分散注意力，=命令会关闭所有的高亮显示效果，直到输入新的搜索模式为止。

左右滚动

如第138页的“左右滚动”一节中所述，可以使用:set nolinewrap来启用左右滚动。与其他编辑器不一样的是，在vile中左右滚动是默认的。过长的行会在左右两端分别标记<与>符号。sideways的值控制vile往右滚动时所移动的字符数。如果设置sideways为0，每一次的滚动将移动1/3个屏幕，否则屏幕即滚动指定的字符数量。

可视模式

vile在高亮显示文本时与elvis和Vim不同。它使用了“引用移动”（quoted motion）命令q。

我们在区域开始的地方输入q，用vi的动作移到区域的结尾处，再输入另一个q结束引用移动。vile会高亮显示这一块标记的文本。

q命令的参数决定了高亮显示的种类。1q（与q同义）表示精确的高亮显示，2q是每次一行的高亮显示，而3q则形成矩形的高亮显示。

引用移动通常与运算符例如d或y一起使用。因此，d3qjjwq将删除一个由移动指示的矩形区域。如果不和运算符一起使用，则只为区域加上高亮显示。区域可用^S引用，因此d ^S可删除高亮显示的区域。

另外，矩形区域可以通过标记（mark）来指示（注11）。各位已经知道，标记可用来引用一个指定的字符（以`引用时）或一个指定的行（以'引用时）。另外，用`b代替'b来引用标记（假设是以mb设置的标记），可以改变这个运算符的本质——d'b将删除一组行，而d`b会删除某两行的部分内容及这两行间的所有行。使用`形式的标记引用可以产生比'形式的标记引用更“精确”的区域。

注11：感谢Paul Fox的解释。

vile增加了第三种的标记引用形式。\命令可以作为另一种引用标记的方式。它本身的作用很像`，会将光标移动到标记被设置的地方。然而如果与运算符结合，作用就不一样了。标记引用范围将变成“矩形”，例如d\b会删除一块矩形区域，区域边角由光标和持有标记b的字符所标记：

按键顺序　　结果

ma

```
The 6th edition of <citetitle>Learning the vi Editor</citetitle>
brings the book into the late 1990&rsquo;s.
In particular, besides the &ldquo;original&rdquo; version of
<command>vi</command> that comes as a standard part of every Unix system,
there are now a number of freely available &ldquo;clones&rdquo;
or work-alike editors.
```

在*book*中的*b*上设置标记a。

3jfr

```
The 6th edition of <citetitle>Learning the vi Editor</citetitle>
brings the book into the late 1990&rsquo;s.
In particular, besides the &ldquo;original&rdquo; version of
<command >vi</command> that comes as a standard part of every Unix system,
there are now a number of freely available &ldquo;clones&rdquo;
or work-alike editors.
```

将光标移到*number*中的*r*上以标记出矩形的另一角。

^A ~\a

```
The 6th edition of <citetitle>Learning the vi Editor</citetitle>
brings the BOOK INTO The late 1990&rsquo;s.
In particulAR, BESIDES the &ldquo;original&rdquo; version of
<command>vi</COMMAND> that comes as a standard part of every Unix system,
there are nOW A NUMBER of freely available &ldquo;clones&rdquo;
or work-alike editors.
```

切换以标记a限制的矩形区域的大小写。

表18-6汇总了定义任意区域并执行操作的命令。

表18-6：vile的块模式命令

命令	操作
q	引用移动的开始与结束
^A r	打开一个矩形
>	将文本往右移动。如果区域是矩形，则与^A r相同
<	将文本往左移动。如果区域是矩形，则与d相同
y	拉动整个区域。vile会记得区域为矩形
c	更改区域。如果是非矩形的区域，会删除所有端点之间的文本，并进入插入模式；若是矩形区域，则会提示要用来填满整行的文本
^A u	将区域中的文本改成大写的
^A l	将区域中的文本改成小写的

表18-6：vile的块模式命令（续）

命令	操作
^A ~	切换区域中字母的大小写
^A SPACE	用空格填满区域
p，P	将文本放回。如果原来的文本是矩形，vile会做矩形的放回
^A p，^A P	强迫先前拉动的文本作为矩形来放回。拉动范围里最长一行的宽度，即成为矩形的宽度

编程辅助

本节讨论vile的编程辅助功能。

编辑－编译的加速

vile用了两个简单直接的vi模式命令管理程序的开发，请见表18-7。

表18-7：vile程序开发的vi模式命令

命令	功能
^X !*command* ENTER	运行*command*，输出结果则保存在名为[Output]的缓冲区
^X ^X	查找下一个错误。vile会解析输出，并移到下一个错误所在的位置

vile能够理解GNU make产生的*Entering directory XXX*与*Leaving directory XXX*消息，甚至可找到位于不同目录中的正确文件。

错误消息会在[Error Expressions]缓冲区中使用正则表达式来解析。vile会自动创建这个缓冲区，并在我们输入^X ^X时予以利用。我们可在需要时加入其他的表达式，而且它有扩展语法，可于错误消息中指定文件名、行号、字段等信息出现的地方。完整的修改细节在在线帮助中有提供，但是可能不需要做什么改变，因为vile已经做得相当好了。

vile的错误查找器也会自动追踪文件的变动，在我们应付每一个错误时，它会自动跟踪内容的新增与删除。

错误查找器会应用通过读入shell命令创建的最新缓冲区上。例如，^X!command会产生一个[Output]缓冲区，而:e !command会产生[!command]缓冲区。错误查找器会适当地予以设置。

你可以使用:error-buffer命令，把错误查找器指向任意一个缓冲区（不只是shell命令的输出）。这可以让你针对先前的编译器或egrep运行结果使用错误查找器。

语法高亮显示

vile支持所有配置中的语法高亮显示。它使用自定义的语法过滤（syntax filter）程序以执行语法着色。过滤程序或许内置在vile里，或许作为外部程序运行。vile通过语法过滤程序传送准备着色的缓冲区内容，读入其加上标记的版本，再应用标记，为缓冲区加上颜色。

注意： 内置过滤程序比外部程序快，且于终端上显示时，可减少shell的干扰。对于某些平台，语法过滤程序可被动态载入，因而使得编辑器的可执行文件缩小，即使不能像内置的过滤程序一样快。

当前共有71种过滤程序以及Unix手册页的专用程序。有些程序不只被用于一种文件。例如，具有类似语法但使用不同关键字的C、C++与Java。

vile提供根据需求运行语法过滤程序的宏，或在我们修改缓冲区时动态过滤。请见表18-8的整理。

表18-8：vile 的语法高亮显示命令

命令	按键组合	功能
:HighlightFilter		在当前的缓冲区上调用语法高亮显示过滤程序。vile 根据缓冲区的扩充特性而选择过滤程序，该特色称为主模式（major mode，请参考第374页的“主模式”一节）。 如果是内置的过滤程序，vile在初始化时即设置自动着色（autocolor）模式，在我们停止修改缓冲区的5秒后调用本宏
:HighlightFilterMsg	^X-q	使用HighlightFilter为当前的缓冲区附上高亮显示。于完成时显示消息
:HighlightClear	^X-Q	清除当前缓冲区的所有高亮显示。这项动作不会改变缓冲区的主模式
:set-highlighting *majormode*		改变缓冲区的主模式为*majormode*并运行语法高亮显示

表18-8：vile 的语法高亮显示命令（续）

命令	按键组合	功能
:show-filtermsgs		显示关于当前缓冲区的语法过滤程序错误消息。如果语法过滤程序找到任何错误，则vile把错误显示在[Filter Messages]缓冲区中，并设置错误缓冲区，让我们能逐一查看找到错误的地方

* 当vile首度于20世纪90年代中期实现语法高亮显示时，表示这项工作的完成还很重要。但时代不同了——机器变快了。

每次运行语法过滤程序，均会读入一份或多份外部文件，文件里包含待高亮显示的关键字以及相应的颜色和显示选项（黑体、下划线、斜体）。程序根据缓冲区的主模式名称搜索这些文件（后缀为.keywords），搜索规则请参考在线帮助。你可以使用:which-keywords宏，以列出vile查找的这类文件的位置以及实际找到的文件的位置。请参考范例 18-1。

范例18-1 ":which-keywords cmode"的输出样本

```
Show which keyword-files are tested for:
        cmode                          ❶
(* marks found-files)

$cwd                                   ❷
  ./.c.keywords
$HOME
  ~/.c.keywords
  ~/.vile/c.keywords
$startup-path                          ❸
* /usr/local/share/vile/c.keywords
```

❶ 主模式名称结尾必为"mode"

❷ 当前的工作目录

❸ vile搜索脚本的路径

无论配置是X11、终端（termcap、terminfo、curses）或Windows，vile的语法过滤程序均使用相同的颜色集，颜色集定义于Action、Comment、Error、Ident、Ident2、Keyword、Keyword2、Literal、Number、Preproc、Type等类里。大多数关键字定义会引用某个类。如此一来，我们只要改变一个文件，就能修改所有配色，通常是 $HOME/.vile.keywords文件。在线帮助文件里对于自定义语法颜色有更详细的说明。

一方面，因为语法高亮显示是以外部程序实现的，它应该可以为不同的程序语言编写任

意数量的高亮显示程序。但另一方面，因为这项功能相当复杂，实在不适合非程序员。关于语法高亮显示的过滤程序，在线帮助介绍了它应该如何运行。

*ftp://invisible-island.net/vile/utilities*目录里包含了用户贡献的过滤程序，可用于为makefile、输入信息、perl、HTML与troff加上颜色。甚至还有一个宏，可以根据RCS文件中各行的日期不同而加上不同颜色！

有趣的功能

本节将介绍vile的许多有趣特色。

vile编辑模型
: vile的编辑模型与vi有些不一样。它以Emacs的概念为基础，提供了按键重新绑定（key rebinding）以及更加动态的命令行。

主模式
: vile支持编辑"模式"。这是几组选项设置，可以方便地编辑不同种类的文件。

过程语言
: vile的过程语言（procedure language）允许我们定义函数与宏，让编辑器更加可编程且更有灵活性。

其他小特色
: vile的许多小特色可以让日常的编辑工作更容易。

vile编辑模型

在vi与其他同类品中，编辑功能已"写死"在编辑器里。命令字符与行动间的关联已经内置在代码中了。例如，按x 键删除字符，而按i键会进入插入模式。如果不用很困难的小技巧，我们不能交换这两个键的功能（如果真的可以交换的话）。

从Emacs到 MicroEMACS，vile发展出的编辑模式（editing mode）与众不同。编辑器里包含已经定义的已有名称的函数，分别执行单一的编辑任务，如delete-next-character或delete-previous-character等等。许多函数都绑定到按键，例如delete-next-character绑定到x（注12）。

vile对其插入内容、命令、选择模式，各有不同风格的按键绑定。接下来说明正常编辑

注12：vile 9.6有421个已定义函数（包括某些只在X11或Win32配置中可用的函数）以及大约260个预定义的按键绑定。

模式的绑定。更改这种绑定关系非常简单，你可以使用:bind-key命令，它的参数则是函数的名称，然后是绑定到函数的按键序列。如稍早所提，你或许需在.vilerc文件中加入以下命令：

```
bind-key incremental-search /
bind-key reverse-incremental-search ?
```

这会将/与?搜索命令改成执行增量搜索。

除了预定义的函数外，vile包含了一个简单的程序语言，可用于撰写过程。如此即可使执行此过程的命令绑定到某个按键序列。GNU Emacs在这方面使用了一种Lisp语言的变体，功能非常强大。vile的版本则比较简单、用途较少。

另外，像Emacs一样，vile命令行非常具交互性。许多命令会显示自己的操作数默认值，如果不对的话还可以修改，或按ENTER键进行选择。当你输入vi模式编辑命令，如更改或删除字符时，可在状态行看到反馈的结果。

Paul在前面所提到的"惊人"ex模式，最明显反映在:s（替换）命令上。它会提示命令的每个部分：搜索模式、替换文本与其他标志。

举个例子，假设你想将文件中所有出现的*perl*改成*awk*。在其他编辑器中，只要输入:1,$s/perl/awk/gENTER，这就是命令行上会出现的东西。下例则显示了你在vile冒号命令行上输入时，会在屏幕上看到的东西：

按键顺序	结果
:1,$s	第一部分的替换命令。
/	`substitute pattern: █` vile会提示要输入搜索模式。任何先前的模式都可以让你重复使用。
perl/	`replacement string: █` 在下一个 / 定界符处，vile会提示输入替换文本。任何先前的文本都可以让你重复使用。
awk/	`(g)lobally, ([1-9])th occurrence on line, (c)onfirm, and/or (p)rint result: █` 在最后一个/定界符处，vile会提示输入可选标志。输入所需的标志后，按下ENTER。

vile在所有适当的ex命令中都会这样做。例如，读入命令（:r）会提示你上一次所读入文件的名称。要再一次读入这个文件，只要单击ENTER。

最后，vile的ex模式命令解析器与其他的编辑器比起来能力较差。例如，你不能用搜索

模式来指定行范围（:/now/,/forever/s/perl/awk/g），且未实现移动命令（m）。实际上，没有实现的功能似乎也不会造成很大的阻碍。

主模式

主模式（major mode）（注13）是一些选项设置的集合，在编辑特定文件类时应用。这些选项有许多都是用在个别缓冲区中，如制表位（tab-stop）设置。

vile提供三种选项类型：

- *universal*，应用至程序
- *buffer*，应用至内存缓冲区的内容
- *window*，应用至窗口（本书用语为“窗格”）

buffer（与*window*）选项设置可为全局或本地值。任何缓冲区（或窗口，依选项而定）均可具有其私有（本地）选项值。如果没有私有值，则使用全局值。主模式在*buffer* 的全局与本地值间多加一个层次，提供没有私有值时缓冲区所使用的选项值。

vile有两个内置的主模式：cmode用来编辑C与C++程序；另外一个是vile mode，用在载入至内存缓冲区的脚本上。采用cmode，你可以用%匹配C预处理器中的条件指示（#if、#else、#endif）。vile会自动依照括号（{与}）位置来做源代码的缩排，另外也会对C的注释做聪明的格式化。tabstop与shiftwidth选项也是根据不同主模式而设置。

使用主模式，你可以将同样的特性应用在不同语言编写的程序中。以下这个Tom Dickey的范例，定义了一个新的主模式shmode，用来编辑Bourne shell script（这对任何Bourne风格的shell，如ksh、bash或zsh都有用）。

```
define-mode sh
set shsuf "\.sh$"
set shpre "^#!\\s*\/.*sh\\>$"
define-submode sh comment-prefix "^\\s*/[:#]"
define-submode sh comments "^\\s*/\\?[:#]\\s+/\\?\\s*$"
define-submode sh fence-if   "^\\s*\\<if\\>"
define-submode sh fence-elif "^\\s*\\<elif\\>"
define-submode sh fence-else "^\\s*\\<else\\>"
define-submode sh fence-fi   "^\\s*\\<fi\\>"
```

shsuf（shell 后缀）变量描述了指示一个文件是shell script的文件名后缀。shpre（shell 前言）变量描述了文件的第一行，该行指出文件中包含了shell代码。接着define-

注13：vile的说明文档中将其拼写为一个单词。

submode命令会增加选项，这些选项只会应用在已经设置对应主模式的缓冲区中。这个范例设置了聪明的注释格式化以及针对shell程序的%命令匹配。

本节的范例比起我们的需求啰嗦得多。vile的脚本语言认得使用~with的较简洁的描述方式：

```
define-mode sh
~with define-submode sh
        suf                     "\.sh$"
        pre                     "^#!\\s*\/.*sh\\>$"
        comment-prefix          "^\\s*/[:#]"
        comments                "^\\s*/\\?[:#]\\s+/\\?\\s*$"
        fence-if                "^\\s*\\<if\\>"
        fence-elif              "^\\s*\\<elif\\>"
        fence-else              "^\\s*\\<else\\>"
        fence-fi                "^\\s*\\<fi\\>"
~endwith
```

在初始化脚本中，vile提供了90种预定义的主模式。请使用:showmajormodes命令查看可用主模式的定义。

在vile把文件读入缓冲区时，它使用suffix与prefix作为决定要应用的主模式的准则（注14）。表18-9列出了所有准则。

表18-9：主模式准则

准则	说明
after	强制已定义主模式的检查顺序在给定主模式后。一般而言，系统依主模式定义顺序而予以检查
before	强制已定义主模式的检查顺序在给定主模式前。一般而言，系统依主模式定义顺序而予以检查
mode-filename (mf)	正则表达式描述用于将设置的对应主模式的文件名称。正则表达式只应用在移除了目录名称后的路径上
mode-pathname (mp)	正则表达式描述用于将设置的对应主模式的路径
preamble (pre)	正则表达式描述用于将设置的对应主模式的文件名称的第一行
qualifiers	说明如何结合preamble与suffixes准则。设置 all，让 vile 同时使用两者；设置any，则为使用其中一项即可
suffixes (suf)	正则表达式描述用于将设置的对应主模式的文件名称后缀。表达式只应用在文件名以第一个点号开始的部分

注14：这些准则是四大类选项、全体计数（counting universal）、缓冲区与窗口。它们并未与表B-5的内容同列，因为设置的方式完全不同。

也可以告诉vile使用指定的主模式，例如：

```
:setl cmode
```

即可设置vile为“c”模式（注15）。但这还不会更新语法高亮显示。请使用宏：

```
:set-h cmode
```

（set-highlighting，参考表18-8）上例即可同时应用主模式与高亮显示。

过程语言

vile的过程语言与MicroEMACS的版本几乎没有差别。注释由分号或双引号开始，环境变量名称（编辑器选项）以$开始，用户的变量名称以%开始。有许多内置的函数，可以当作比较与测试的条件，其名称都以&开始。流程控制命令与某些命令以~开始。@ 加上字符串会提示用户输入内容，并返回用户的答案。以下是个出自macros.doc的范例，有点异想天开，但应该能让大家体会一下这种语言的风格：

```
~if &sequal %curplace "timespace vortex"
        insert-string "First, rematerialize\n"
~endif
~if &sequal %planet "earth"     ;If we have landed on earth...
        ~if &sequal %time "late 20th century"  ;and we are then
                write-message "Contact U.N.I.T."
        ~else
                insert-string "Investigate the situation....\n"
                insert-string "(SAY 'stay here Sara')\n"
        ~endif
~elseif &sequal %planet "luna"  ;If we have landed on our neighbor...
        write-message "Keep the door closed"
~else
        setv %conditions @"Atmosphere conditions outside? "
        ~if &sequal %conditions "safe"
                insert-string &cat "Go outside......" "\n"
                insert-string "lock the door\n"
        ~else
                insert-string "Dematerialize..try somewhen else"
                newline
        ~endif
~endif
```

你可以将这些过程存储在某个编号的宏中，或者为其命名以便绑定到按键。上面这个过程在使用Tardis的vile移植时特别有用。

接下来提供较为符合现实情况的范例，Paul Fox运行了grep，在所有C源文件中搜索光标

注15：setl命令设置缓冲区的本地特性。如果vile不能认出文件，:set cmode即设置默认的主模式。

所在位置的单词，接着将结果放在以此单词命名的缓冲区中，并设置让内置的错误查找器（^X ^X）使用这个输出结果作为查找器访问的行列表。最后，将此宏绑定到^A g。~force命令会让后面的命令失效，但不产生错误消息：

```
14 store-macro
        set-variable %grepfor $identifier
        edit-file &cat "!egrep -n " &cat %grepfor " *.[ch]"
        ~force rename-buffer %grepfor
        error-buffer $cbufname
~endm
bind-key execute-macro-14 ^A-g
```

用户定义的过程可有参数，这点很像Bourne shell——但参数能被限制为指定数据类型。如此可让过程与vile的编辑模型（与命令历史记录机制）如预期般运作。但过程不能完全与内置命令互换，因为没有机制可让撤销（undo）功能把整个宏视为单一操作。

最后一点，read-hook与write-hook变量分别可以设为在读入文件之后与写入文件之前要运行的过程的名称。这可以让你做到类似elvis中操作前与操作后命令文件以及Vim中自动命令的功能。

这个语言具有足够的功能，包括流程控制与比较功能以及可以访问许多vile内部状态的变量。vile发行包的macros.doc文件中描述了这个语言的细节。

其他小特色

几个值得一提的小特色：

让vile从管道接受输入

如果让vile成为管道（pipeline）中最后一个命令，则会产生一个名为[Standard Input]的缓冲区且系统会编辑此缓冲区。这也许就是“分页器的终结者”。

编辑Windows文件

将dos选项设为true时，会让vile在读入文件时把一行结尾的换行符忽略掉，而在写入时恢复。这使得在Unix或GNU/Linux系统中编辑Windows文件更方便。

文本重新格式化

^A f命令会重新格式化文本，对选中的文本执行自动换行（word wrapping）。它可以区分出C与shell的注释（以*或#开头）和引用的电子邮件（以>开头）。它类似Unix的fmt命令，但速度比较快。

信息行的格式化

modeline-format变量是一个字符串，用于控制vile格式化状态行的方式。状态行是每个窗口最底下的一行，用于描述缓冲区的状态，如名称、当前的主模式、修改

状态、插入或命令模式等等（注16）。

这个字符串包含了*printf*(3) 风格的百分号序列。例如，%b代表缓冲区名称；%m表示主模式；如果设置了ruler，则%l表示行编号。非格式化的字符会照原样输出。

vile还有许多其他特性。熟练使用vi后，改用vile会很容易。vile的可编程能力提供了灵活性，而其本质上的交互性以及默认值的使用，使得其比传统的vi来得友善。

资源与支持的操作系统

vile的官方下载地址是在*http://invisible-island.net/vile/vile.html*，其ftp下载地址是*ftp://invisible-island.net/vile/vile.tar.gz*。vile.tar.gz文件是指向当前版本的符号链接。

vile是以ANSI C撰写的。它也可以在Unix、OpenVMS、MS-DOS、Win32、Win32 控制台与Win32 GUI、BeOS、QNX、OS/2上构建与运行。

编译vile的过程非常简单。通过ftp或网页取得发行包，将其解压缩并逐个解开文件之后，运行configure程序，然后运行make：

```
$ gzip -d < vile.tar.gz | tar -xvpf -
...
$ cd vile-8.0; ./configure
...
$ make
...
```

vile应该就会顺利配置并编译程序。使用make install来安装它。

注意： 如果想让语法着色的运作顺畅，你或许希望在运行configure时加上--with-builtin-filters选项。应该使用flex（2.54a 或更新版本），而不要使用lex，因为这项工具的 Unix 版本的运作不太好。configure脚本也不兼容太旧的flex版本。

如果需要报告vile的错误或问题，可以发送电子邮件到*vile@nongnu.org*。这是比较好的方式。如果必要的话，也可以直接联络Tom Dikey，其电子邮件地址为*dickey@invisible-island.net*。

注16：vile的说明文档以*modeline*称呼状态行。然而，vile中也实现了vi modeline 功能，故我们还是使用状态行这种称呼，以免混淆。

第四部分

附录

第四部分提供vi用户应该会有兴趣的参考资料。该部分包含下列附录：

附录A，*vi、ex与Vim编辑器*

附录B，*设置选项*

附录C，*问题集*

附录D，*vi与国际互联网*



附录A

vi、ex与Vim编辑器

本章附录整理了vi快速引用格式下的标准功能。涵盖了利用冒号输入的命令（也是大家所熟知的ex命令，因为它们的开发可追溯到编辑器的源头），还有最受欢迎的Vim功能。

本章附录有如下主题：

- 命令行语法
- vi操作的复习
- 依字母顺序排列的命令模式中的按键（组合）列表
- vi命令
- vi配置
- ex基础
- ex命令汇总（依字母顺序排列）

命令行语法

以下是最常见的三种vi会话开启方式：

```
vi [options] file
vi [options] +num file
vi [options] +/pattern file
```

我们可以单纯打开编辑用的*file*、打开时选择的光标位置（第*num*行）或打开后跳到第一次出现*pattern*的行。如果没有指定*file*，vi将打开空缓冲区。

命令行选项

因为vi与ex是相同的程序，故它们共享相同选项。然而，有些选项只在其中一个程序里才有合理的作用。Vim专用的选项另有标记：

+[*num*]

编辑工作从指定的*num*行开始；如果省略了*num*，则于最后一行开始。

+/*pattern*

编辑工作从匹配*pattern*的第一行开始（在ex下，启动文件.exrc中若设置了nowrapscan，这个选项会失效，因为ex会从文件的最后一行开始编辑）。

+?*pattern*

编辑工作从匹配*pattern*的最后一行开始。

-b

以二进制模式编辑文件。{Vim}

-c *command*

在启动后立即运行给定的ex命令。vi只允许使用一个-c选项，Vim则可接受10个-c选项。本选项的旧形式+*command*仍得到支持。

--cmd *command*

与-c功能很像，但在读入任何源文件前执行命令。{Vim}

-C

Solaris vi：与-x相同，但假设文件已被加密。

Vim：以兼容vi的模式启动编辑器。

-d

以diff模式运行。运作方式接近vimdiff。{Vim}

-D

使用脚本时的调试模式。{Vim}

-e

如ex般运行（单行编辑而非满屏模式）。

-h

打印帮助消息，然后离开。{Vim}

-i *file*

保存或恢复Vim状态时使用指定*file*而非默认值（~/.viminfo）。{Vim}

-l

进入Lisp模式以运行Lisp程序（并非所有版本均支持）。

-L

列出因为中止编辑会话或系统死机而保存的文件（并非所有版本均支持）。在Vim中，这个选项的功能与-r相同。

-m

启动编辑器时关闭write选项，用户将不能写入文件。{Vim}

-M

不允许修改文件中的文本。{Vim}

-n

不使用swap文件，只在内存中记录改变。{Vim}

--noplugin

不载入任何插件程序。{Vim}

-N

以不兼容vi的模式运行Vim。{Vim}

-o[*num*]

启动Vim时打开*num*个窗口。默认为每个文件打开一个窗口。{Vim}

-O[*num*]

启动Vim时打开*num*个水平排列（垂直分割）的窗口。{Vim}

-r[*file*]

恢复模式（recovery mode）。在中止编辑会话或系统死机后，可恢复并继续*file*的编辑。未指定*file*时，则列出可供恢复的文件。

-R

于只读模式中编辑文件。

-s

安静的（silent）不出现提示消息。这在运行脚本时很有用。这项行为也能通过较旧的-选项来设置。在Vim中，本选项只能与-e一同使用。

-s *scriptfile*

读入并执行指定的*scriptfile*里给定的命令，把它们当成来自键盘的输入。{Vim}

-S *commandfile*

在载入命令行中指定的任何欲编辑文件后，读入并执行给定的*commandfile*里的命令。是vim -c 'source *commandfile*' 的缩写。{Vim}

`-t` *tag*

编辑包含*tag*的文件，并把光标置于*tag*的位置。

`-T` *type*

设置可选终端类型。这个值将覆盖环境变量$TERM。{Vim}

`-u` *file*

从指定的源文件中读入配置信息，而不是从默认的`.vimrc`源文件中读入。如果*file*参数为`NONE`，Vim将不会读入源文件、不会载入插件程序并于兼容模式中运行；如果*file*参数为`NORC`，Vim将不会读入源文件，但会载入插件程序。{Vim}

`-v`

以满屏模式运行（vi的默认值）。

`--version`

打印版本信息，然后离开。{Vim}

`-V[`*num*`]`

详细模式（verbose mode）。打印被设置的选项以及被读/写的文件信息。可设置信息的详尽程度，以增减收到的消息量。默认值为10，表示非常详尽。{Vim}

`-w` *rows*

设置窗口尺寸，使得一次呈现*rows*行，这在通过低速拨号网络（或通过远距离国际互联网连接）编辑时很有用。较旧版本的`vi`不允许在选项和参数间加上空格。Vim不支持此选项。

`-W` *scriptfile*

从当前的会话中把所有的键入命令写入指定的*scriptfile*。文件可用`-s`命令创建。{Vim}

`-x`

提示请求使用`crypt`加密或解密文件的按键（并非所有版本均支持）（注1）。

`-y`

无模式的`vi`。只以插入模式运行Vim，没有命令模式。与调用Vim作为`evim`相同。{Vim}

`-Z`

于限制模式（restricted mode）中启动Vim。不允许使用shell命令或暂停编辑器。{Vim}

注1：crypt命令的加密很脆弱，别用来保护重大秘密。

虽然大多数人都因vi的运用而知道ex命令，但ex本身也是个独立的程序，可从shell中调用（例如编辑文件作为脚本的一部分）。在ex中，可以输入vi或visual命令以启动vi。相似地，在vi中，可以输入Q以离开vi编辑器并进入ex。

离开ex的方式如下所示：

:x　离开（保存改变并离开）。

:q!　离开而不保存改变。

:vi　进入vi编辑器。

复习vi的操作

本节复习下列主题：

- vi模式
- vi命令的语法
- 状态行命令

命令模式

打开文件后，随即进入命令模式（command mode）。在命令模式中，可以：

- 调用插入模式（insert mode）
- 下达编辑命令
- 移动光标至文件中的不同位置
- 调用ex命令
- 调用 Unix shell
- 保存文件的当前版本
- 离开vi

插入模式

在插入模式中，可在文件中输入新的文本。一般以i命令进入插入模式。按下ESC键可离开插入模式，回到命令模式。可进入插入模式的命令的完整列表，请参考第391页的“插入命令”一节。

vi命令的语法

vi中，编辑命令具有如下形式：

```
[n] operator [m] motion
```

基础编辑*operator*（运算符）包括：

c　　开始修改。

d　　开始删除。

y　　开始拖动（或复制）。

如果当前的行是编辑操作的对象，*motion*部分则与运算符相同：cc、dd、yy。否则，编辑运算符将作用在光标移动命令指定的对象或模式匹配命令指定的对象上（例如，cf.改变至下一个点号的位置）。*n*与*m*表示编辑操作执行的次数，或是应用编辑操作的对象数量。如果*n*与*m*都被指定，编辑效果将为$n \times m$。

编辑操作的对象可为下列任何文本块：

词

　　范围包括到空白字符（空格或 tab）前或标点符号前的字符。大写对象是一种变体，只能识别空白。

句子

　　范围到.、!或?为止，后接两个空格。

段落

　　范围到下一个空白行或para=选项定义的段落宏为止。

小节

　　范围到由sect=选项定义的下一个nroff/troff节标题为止。

移动

　　范围到移动指示符指定的字符或其他文本对象为止，包括模式搜索指定的对象。

范例

2cw　　改变接下来的两个词。

d}　　删除到下个段落为止。

d^　　向后删除到该行的开头为止。

5yy　　复制接下来的5行。

y]]　　复制到下个小节为止。

cG　　改变到编辑缓冲区的结尾处为止。

稍后在本附录的“改变与删除文字”小节中，还有更多命令与范列。

可视模式（Vim）

Vim提供了一种额外模式——“可视模式”（visual mode）。这种模式能高亮显示的文本块，该块是编辑命令（如删除、保存或拖动）的操作对象。Vim的图形化版本可让我们使用鼠标来以相近的方式高亮显示文本。请参考第172页的“可视模式的移动”一节以了解更多相关信息。

v　　在可视模式中一次选择一个字符。

V　　在可视模式中一次选择一行。

CTRL-V　　在可视模式中选择文本块。

状态行命令

大多数命令在我们输入时不会回显在屏幕上。但在屏幕底端的状态行可用于编辑这些命令：

/　　向前搜索模式。

?　　向后搜索模式。

:　　调用ex命令。

!　　调用Unix命令，这个命令以缓冲区中的对象作为其输入，并把对象替换为命令的输出。在!后键入移动命令来描述传送给Unix命令的对象。请于状态行键入命令。

要把命令键入状态行，还需按下ENTER键。另外，错误消息与CTRL-G命令的输出内容也会显示在状态行上。

vi命令

vi在命令模式中支持许多单键命令。Vim则额外支持多键命令。

移动命令

有些版本的vi不能识别扩展的键盘按键（例如箭头、page up、page down、home、insert、delete），也有些能识别。不过，所有版本都能识别本节提到的按键。多数vi用

户倾向于使用这些按键，因为有助于让手指停留在键盘上的固定位置。在命令前加上数值，则可重复动作。移动命令也能用在运算符后。运算符将被应用在待移动的内容上。

字符

h、j、k、l	左、下、上、右（←、↓、↑、→）
空格键	右
BACKSPACE	左
CTRL-H	左

文本

w、b	向前或向后一个“词”（字母、数字与下划线构成词）。
W、B	向前或向后一个“词组”（只包括以空白分隔的内容）。
e	词的结尾处。
E	词组的结尾处。
ge	前一个词的结尾处。{Vim}
gE	前一个词组的结尾处。{Vim}
)、(	下一个或当前句子的开头处。
}、{	下一个或当前段落的开头处。
]]、[[	下一个或当前小节的开头处。
][、[]	下一个或当前小节的结尾处。{Vim}

行

文件中较长的行或许会在屏幕上显示为多行（从屏幕上的一行绕排到下一行）。虽然大多数命令依照文件里定义的行而运作，但有些命令却依照屏幕上的行而运作。Vim 的wrap选项能控制一行显示的长度。

0、$	当前这行的第一个或最后一个位置。
^、_	当前这行的第一个非空格字符。
+、-	下一行或前一行的第一个非空格字符。
ENTER	下一行的第一个非空格字符。
num |	当前这行的第*num*栏。
g0、g$	屏幕所见一行的第一个或最后一个位置。{Vim}
g^	屏幕所见一行的第一个非空格字符。{Vim}

gm	屏幕所见一行的中点。{Vim}
gk、gj	向上或向下移动屏幕所见的一行。{Vim}
H	屏幕上最顶端的一行（Home的位置）。
M	屏幕上的中间一行。
L	屏幕上的最后一行。
num H	从最顶行往下数的第*num*行。
num L	从最底行往上数的第*num*行。

屏幕

CTRL-F、CTRL-B	向前或向后滚动一个屏幕。
CTRL-D、CTRL-U	向下或向上滚动半个屏幕。
CTRL-E、CTRL-Y	向屏幕顶端或底端多滚动一行。
z ENTER	把光标所在的行移到屏幕顶端。
z.	把光标所在的行移到屏幕中间。
z-	把光标所在的行移到屏幕底端。
CTRL-L	重绘屏幕（不做滚动）。
CTRL-R	vi:重绘屏幕（不做滚动）。 Vim：重做最后一次恢复的改变。

搜索

/ *pattern*	向前搜索*pattern*。按下ENTER表示搜索结束。
/ *pattern* /+ *num*	转至*pattern*后的第*num*行。向前搜索*pattern*。
/ *pattern* /- *num*	转至*pattern*前的第*num*行。向前搜索 *pattern*。
? *pattern*	向后搜索*pattern*。按下ENTER表示搜索结束。
? *pattern* ?+ *num*	转至*pattern*后的第*num*行。向后搜索*pattern*。
? *pattern* ?- *num*	转至*pattern*前的第*num*行。向后搜索*pattern*。
:noh	暂停搜索的高亮显示，直到下一次搜索。{Vim}。
n	重复前一次搜索。
N	朝反方向重复搜索。
/	向前重复前一次搜索。
?	向后重复前一次搜索。
*	向前搜索光标所在处的词。需匹配出完全相符的词。{Vim}

#	向后搜索光标所在处的词。需匹配出完全相符的词。{Vim}
g*	向前搜索光标所在处的词。可匹配出嵌在较长词里的单词的字符。{Vim}
g#	向后搜索光标所在处的词。可匹配出嵌在较长词里的单词的字符。{Vim}
%	搜索与当前的圆括号、花括号、方括号成对的符号。
f*x*	向前移动光标至当前行的*x*位置。
F*x*	向后移动光标至当前行的*x*位置。
t*x*	向前移动光标至当前行里*x*处的前一个字符。
T*x*	向后移动光标至当前行里*x*处的后一个字符。
,	搜索与前一次f、F、t或T相反的方向。
;	重复前一次 f、F、t或T。

行编号

CTRL-G	显示当前的行编号
gg	移动至文件的第一行。{Vim}
num G	移动至指定的行编号*num*。
G	移动至文件的最后一行。
: *num*	移动至指定的行编号*num*。

标记

m*x*	在当前的位置加上标记*x*。
`*x*	（反引号）移动光标至标记*x*。
'*x*	（单引号）移动至包含标记*x*的行的开始处。
``	（反引号）回到最近一次移动前的位置。
''	（单引号）与上一项相似，但回到某行的开始处。
'"	（单引号、双引号）移动到最后一次编辑该文件的位置。{Vim}
`[, `]	（反引号、方括号）移动到前一次文本操作的开始处／结尾处。
'[, ']	（单引号、方括号）与上一项相似，但回到被操作的文本行的开始处。{Vim}
`.	（反引号、点号）移动到文件前次改变的位置。{Vim}
'.	（单引号、点号）与上一项相似，但回到改变的行的开始处。{Vim}
'0	（单引号、零）位置设于上次离开Vim的地方。{Vim}

:marks　　　列出活动中的标记。{Vim}

插入命令

a　附加到光标后。

A　附加到一行的结尾处。

c　开始改变内容的操作。

C　于一行结尾处开始改变内容。

gI　于一行开始处插入。{Vim}

i　在光标前的位置插入。

I　在一行的开始处插入。

o　在光标下一行开始一行。

O　在光标上一行开始一行。

R　开始覆盖文本。

s　替换一个字符。

S　替换整行。

ESC　终止插入模式。

下列命令可于插入模式中运作：

BACKSPACE　删除前一个字符。

DELETE　删除当前的字符。

TAB　插入一个tab。

CTRL-A　重复上一次插入。{Vim}

CTRL-D　整行向左移动一个shiftwidth的距离。{Vim}

CTRL-E　插入位于光标下一行的字符。{Vim}

CTRL-H　删入前一个字符（与退格键相同）。

CTRL-I　插入一个tab。

CTRL-K　开始插入多按键字符。

CTRL-N　在光标左侧插入下一个模式的自动补全。{Vim}

CTRL-P　在光标左侧插入前一个模式的自动补全。{Vim}

CTRL-T　整行向右移动一个shiftwidth的距离。{Vim}

CTRL-U　删除当前行。

CTRL-V　逐个插入下个字符。

CTRL-W	删除前一个词。
CTRL-Y	插入光标处的字符。{Vim}
CTRL-[	与ESC相同。终止插入模式。

上表所示的控制字符有些由stty设置。你的终端设置可能与我们的不同。

编辑命令

请回想一下，基础编辑运算符就是c、d与y。

改变与删除文本

下表虽不算巨细靡遗，但勾勒出最常用的操作：

cw	改变一个词。
cc	改变一行。
c$	改变从当前的位置到行尾的文本。
C	与c$相同。
dd	删除当前行。
num dd	删除*num*行（*num*是数量）。
d$	删除从当前的位置到行尾的文本。
D	与d$相同。
dw	删除一个词。
d}	删除至下个段落。
d^	向后删除至该行的开始处。
d/ *pat*	删除至模式*pat*首度出现处。
dn	删除至模式*pat*下次出现处。
df *x*	删除至当前行中*x*所在的位置（包括*x*）。
dt *x*	删除至当前行中*x*所在的位置（但不包括*x*）。
dL	删除至屏幕上的最后一行。
dG	删除至文件的结尾处。
gqap	根据textwidth重新格式化当前的段落。{Vim}
g~w	转换词的大小写。{Vim}
guw	改变词为全部小写的。{Vim}
gUw	改变词为全部大写的。{Vim}
p	插入上一个删除或移动的文本到光标后。

gp	与p相同，但把光标放在被插入的文本后。{Vim}
gP	与P相同，但把光标放在被插入的文本后。{Vim}
]p	与p相同，但符合当前的缩排。{Vim}
[p	与P相同，但符合当前的缩排。{Vim}
P	插入上一个删除或移动的文本到光标前。
r*x*	以*x*取代字符。
R *text*	以新的*text*取代（覆盖），从光标位置开始。ESC结束取代模式。
s	替换字符。
4s	替换4个字符。
S	替换整行。
u	撤销最后一次改变。
CTRL-R	重做最后一次改变。{Vim}
U	恢复当前行。
x	删除当前光标位置处的字符。
X	后退删除一个字符。
5X	删除光标前的5个字符。
.	重复前一次改变。
~	颠倒大小写并把光标右移。
CTRL-A	递增光标下的数字。{Vim}
CTRL-X	递减光标下的数字。{Vim}

复制与移动

寄存器（register）名称可用字母a到z。大写名称是把文本内容附加到相应的寄存器里。

Y	复制当前行。
yy	复制当前行。
" *x* yy	复制当前行到寄存器*x*。
ye	复制文本到词尾。
yw	与ye一样，但在词后包括空格。
y$	复制该行的剩下部分。
" *x* dd	删除当前行至寄存器*x*。
"*x*d	删除至寄存器*x*。
"*x*p	粘贴寄存器*x*的内容。

y]]	复制到下个小节标题为止。
J	合并当前行与下一行。
gJ	与J相同，但不会插入空白。{Vim}
:j	与J相同。
:j!	与gJ相同。

保存与离开

写入文件表示以当前的文本覆盖指定的文件。

ZZ	离开vi，只在有改变时写入文件。
:x	与ZZ相同。
:wq	写入文件并离开。
:w	写入文件。
:w *file*	保存副本至文件（*file*）。
: *n* , *m* w *file*	把第*n*行到第*m*行写入新文件（*file*）。
: *n* , *m* w >> *file*	把第*n*行到第*m*行附加至现有的文件（*file*）。
:w!	写入文件（覆盖保护）。
:w! *file*	以当前的文本覆盖文件（*file*）。
:w %. *new*	把名为*file*的当前缓冲区中写入为*file.new*。
:q	离开vi（若已改变文件则操作失效）。
:q!	离开vi（丢弃编辑内容）。
Q	离开vi并调用ex。
:vi	在Q命令后返回到vi。
%	在编辑命令中将被替换为当前文件名。
#	在编辑命令中将被替换为候补文件名。

访问多个文件

:e *file*	编辑另一个文件（*file*），当前文件成为候补。
:e!	返回到当前文件前次写入的版本。
:e + *file*	于文件（*file*）结尾处开始编辑。
:e +*num* *file*	打开文件（*file*）后直接到第*num*行。
:e #	打开候补文件，直接到先前编辑的位置。

:ta *tag*	于标签（*tag*）的位置编辑文件。
:n	编辑文件列表中的下个文件。
:n!	强制编辑下个文件。
:n *files*	指定新的文件（*files*）列表。
:rewind	编辑列表中的第一个文件。
CTRL-G	显示当前的文件与行编号。
:args	列出预备编辑的文件的列表。
:prev	编辑文件列表中的前一个文件。

窗口命令（Vim）

下表列出了Vim中用于控制窗口的常见命令。也可见第400页的“ex命令汇总（依字母顺序排列）”一节，参考split、vsplit、resize等命令。为了简洁起见，本表把控制字符标记为^。

:new	打开一个新窗口。
:new *file*	于一个新窗口打开*file*。
:sp [*file*]	分割当前的窗口。加入*file*时，于新窗口中打开指定文件。
:sv [*file*]	与:sp相同，但新窗口限定为只读的。
:sn [*file*]	于新窗口中编辑文件列表中的下一个文件。
:vsp [*file*]	与:sp类似，但垂直分割窗口，而非水平分割。
:clo	关闭当前窗口。
:hid	隐藏当前窗口，除非它是唯一可见的窗口。
:on	使当前窗口为唯一可见的窗口。
:res *num*	调整窗口尺寸为*num*行。
:wa	把所有改变了的缓冲区写入它们的文件。
:qa	关闭所有缓冲区并离开。
^W s	与:sp相同。
^W n	与:new相同。
^W ^	于新窗口中打开（稍早编辑的）候补文件。
^W c	与:clo相同。
^W o	与:only相同。
^W j、^W k	移动光标至下一个／前一个窗口。
^W p	移动光标至前一个窗口。

^W h、W l	移动光标至屏幕左侧／右侧的窗口。
^W t、W b	移动光标至屏幕顶端／底端的窗口。
^W K、W B	移动当前窗口至屏幕顶端／底端。
^W H、W L	移动当前窗口至屏幕左侧／右侧。
^W r、W R	向下／上轮替窗口。
^W +、W -	递增／递减当前窗口的尺寸。
^W =	平均分配所有窗口的高度。

与系统交互

:r *file*	在光标后读入*file*的内容。
:r !*command*	在当前行后读入*command*的输出。
: *num* r !*command*	类似上一项，但输出结果放在第*num*行后（0表示放在文件顶端）。
:!*command*	运行*command*，然后返回。
!*motion command*	把*motion*涵盖的文本传送给Unix *command*，以命令的输出替换文本。
: *n* , *m* !*command*	传送第*n*到*m*行给*command*，以命令的输出替换文本。
num!!*command*	传送*num*行给Unix *command*，以命令的输出替换文本。
:!!	重复上一个系统命令。
:sh	创建subshell，以*EOF*返回给编辑器。
CTRL-Z	暂停编辑器，用fg恢复。
:so *file*	从*file*读入并执行ex命令。

宏

:ab *in out*	使用*in*作为*out*在插入模式中的缩写。
:unab *in*	移除缩写*in*。
:ab	列出缩写。
:map *string sequence*	映射*string*至一组命令*sequence*。例如使用#1、#2表示功能键等等。
:unmap *string*	移除对*string*的映射。
:map	列出已被映射到命令的*string*。
:map! *string sequence*	映射*string*至插入模式中的一组按键*sequence*。

`:unmap! string`	移除插入模式中的映射（或许需以CTRL-V括起字符）。
`:map!`	列出用于插入模式的映射的字符串。
`qx`	以字母x指定寄存器，于该寄存器中记录键入的字符。如果字母为大写的，则字符附加至寄存器内容。{Vim}
`q`	停止记录。{Vim}
`@x`	执行以字母x指定的寄存器。使用@@重复前一个 @ 命令。

在vi中，下列字符不会用在命令模式中，能被映射为用户定义的命令：

字母

g、K、q、V、v

控制键

^A、^K、^O、^W、^X、^_、^\

符号

_、*、\、=、#

注意： 如果设置了Lisp模式，则=被vi使用。不同版本的vi或许会使用一些上述字符，最好在使用前先做测试。

Vim中不能使用^K、^_、_、\。

其他命令

<	把接下来的移动命令涵盖的文本向左移动一个shiftwidth。{Vim}
>	把接下来的移动命令涵盖的文本向右移动一个shiftwidth。{Vim}
<<	整行向左移动一个shiftwidth（默认为8个空格）。
>>	整行向右移动一个shiftwidth（默认为8个空格）。
>}	向右移动到段落结尾。
<%	向右移动到成对的圆括号、花括号、方括号为止（光标必须放在所需对应的符号上）。
==	以C语言的风格缩排文本行，或使用equalprg选项中指定的程序。
g	于Vim中开始许多多字符命令。
K	在手册页（或定义于keywordprg中的程序）中寻找光标处的词。{Vim}
^O	回到前一次跳到的位置。{Vim}
^Q	与^V相同。{Vim}（在某些终端上则会恢复数据流动）。

^T	回到标签栈中的前一个位置（Solaris vi、Vim、nvi、elvis、vile）。
^]	对光标下的文本执行标签查找。
^\	进入ex行编辑模式。
^^	（同时按下Ctrl键与^键）回到先前编辑的文件。

vi配置

本节内容包括：

- :set命令
- 能以:set取得的选项
- .exrc文件范例

:set命令

:set命令允许我们具体指定改变编辑环境特性的选项。选项或许放在~/.exrc文件里，或于vi会话中设置。

如果命令放在.exrc里，可以不用输入冒号：

:set *x*	启用布尔类型的选项*x*；显示其他类型选项的值。
:set no *x*	禁用选项*x*。
:set *x* = *value*	将*value*给予选项*x*。
:set	列出改变的选项。
:set all	列出所有选项。
:set *x* ?	列出选项*x*的值。

附录B提供了Solaris vi、Vim、nvi、elvis与vile的:set选项列表。请参考附录B以获得更多信息。

.exrc文件范例

在ex脚本文件中，注释以双引号字符开始。下列各行代码是自定义.exrc文件的范例：

```
set nowrapscan                    " 搜索到达文件结尾时不绕回文件开头
set wrapmargin=7                  " 在距离右边界7栏时绕排文本
set sections=SeAhBhChDh nomesg    " 设置troff宏，不允许显示消息
map q :w^M:n^M                    " 移动至下一个文件的别名处
map v dwElp                       " 移动一个词
```

```
ab ORA O'Reilly Media, Inc.        " 输入缩写
```

注意： Vim不需要q作为别名，它有:wn命令。别名v将可能隐藏Vim的v命令，v命令用于进入一次输入一个字符的可视模式操作。

ex基础

ex行编辑器是vi满屏编辑器的基础。ex命令运作在当前行或运作在一个范围的行上。最常在vi里使用ex。在vi中，需在ex命令前加上冒号，然后按下ENTER输入命令。

也可以调用ex——从命令行，与调用vi的方式一样（亦可以相同方式执行ex脚本）。或可使用vi命令Q离开vi编辑器并进入ex。

ex命令的语法

欲在vi中输入ex命令，请输入：

```
:[address] command [options]
```

初始的:表示ex命令。在输入命令时，它会回显在状态行上。按下ENTER键即执行命令。*address*是*command*执行对象的行编号或范围。稍后另有*options*与*address*的说明。ex命令则在第400页的“ex命令汇总（依字母顺序排列）”一节另有讨论。

离开ex的方式包括：

:x	离开（保存改变并离开）。
:q!	不保存改变就离开。
:vi	切换到vi编辑器，编辑当前的文件。

地址

如果没有给定地址（address），当前行就是命令执行的对象。如果地址指定为一个行范围，其格式为：

x,y

*x*与*y*分别是地址里的第一行与最后一行（*x*在缓冲区里的位置必须比*y*前）。*x*与*y*可能是行编号或符号。使用;代替,，可把当前行设置为*x*，再解释*y*。1, $表示法表示文件中的每一行，与%相同。

地址符号

`1,$`	文件中的所有行。
x,y	从*x*行到*y*行。
x;*y*	从*x*行到*y*行，当前行复位为*x*。
`0`	文件顶端。
`.`	当前行。
num	绝对行编号*num*。
`$`	最后一行。
`%`	所有行，与`1,$`相同。
x-n	*x*前的*n*行。
x+n	*x*后的*n*行。
-[*num*]	1或*num*行前。
+[*num*]	1或*num*行后。
'*x*	（单引号）标记为*x*的行。
''	（两个单引号）前一个标记处。
/*pattern*/	前进到匹配*pattern*的行。
?*pattern*?	后退到匹配*pattern*的行。

请参考第六章以了解模式的使用细节。

选项

!

指示命令的变体形式，覆盖正常行为。!必须直接出现在命令后。

count（数量）

命令重复的次数。不像vi命令中的使用方式，*count*不能放在命令前，因为ex命令前的数字会被当成行地址。例如，d3表示删除3行，从当前行开始；3d则表示删除第3行。

file（文件）

受命令影响的文件的名称。%表示当前文件，#表示前一个文件。

ex 命令汇总（依字母顺序排列）

ex命令可借由指定任意独特的缩写来输入。在接下来的参考条目中，全名采用小标题般

的粗黑字体，可用的最短缩写则列在其下的语法示范中。本节范例均假设通过vi输入，所以会加上:提示符。

abbreviate

ab [*string text*]

定义*string*，当输入时会被翻译成*text*。如果并未指定*string*与*text*，则列出当前的所有缩写。

范例

注意：输入^V后按下ENTER，即出现^M。

```
:ab ora O'Reilly Media, Inc.
:ab id Name:^MRank:^MPhone:
```

append

[*address*] a[!]
text
.

附加新*text*于指定*address*；若未指定，则附加于当前地址。加上!可切换在输入期间使用的autoindent设置。也就是说，如果启用了autoindent，! 将禁用它。键入命令后再键入新文本。终止输入文本的方式，是键入只包含一个点号的行。

范例

```
:a                        开始附加至当前的行
Append this line
and this line too.
.                         终止附加输入的文本
```

args

ar
args *files* ...

列出参数列表的成员（指名于命令行上的文件），列出当前的参数时加上方括号([])。

第二组语法为Vim使用，能让你重设预备编辑的文件的列表。

bdelete

[*num*] bd[!] [*num*]

卸载缓冲区*num*并从缓冲区列表中将其删除。加上!强制移除未保存的缓冲区。缓冲区也能用文件名称指定。如果没有指定缓冲区，则移除当前的缓冲区。{Vim}

buffer

[*num*] b[!] [*num*]

开始编辑缓冲区列表中的缓冲区*num*。加入!强制切换离开未保存的缓冲区。缓冲区也能以文件名称指定。如果没有指定缓冲区，则继续当前的编辑操作。{Vim}

buffers

buffers[!]

列出缓冲区列表的成员。有些缓冲区（例如删除缓冲区）不会被列出。加上!可呈现非列出的缓冲区。ls是这个命令的另一种缩写。{Vim}

cd

cd *dir*
chdir *dir*

在编辑器里改变当前目录为*dir*。

center

[*address*] ce [*width*]

把行放在指定*width*中间。如果未指定*width*，则使用textwidth。{Vim}

change

[*address*] c[!]
text
.

以*text*取代指定的行。加上!可于*text*输入期间调换autoindent的设置。终止输入的方式，是输入只包含一个点号的行。

close

```
clo[!]
```

关闭当前的窗口，除非它是最后一个窗口。如果窗口里的缓冲区并未在另一个窗口中打开，则从内存中将其卸载。这个命令不会关闭尚有未保存改变的缓冲区，但加入!后，改为隐藏该缓冲区。{Vim}

copy

[*address*] co *destination*

复制包括在*address*里的行到指定的*destination*（目的）地址。命令t（"to"的缩写）是copy的同义词。

范例

:1,10 co 50	*把最前面的10行复制到第50行后*

delete

[*address*] d [*register*] [*count*]

删除包括在*address*里的行。如果指定了*register*（寄存器），则保存文本或把文本附加到指定的寄存器里。寄存器名称为小写的a到z。大写的名称则附加文本到相应的寄存器。如果指定了*count*，则删除指定数量的行。

范例

:/Part I/,/Part II/-1d	*删除到"Part II"之前的行*
:/main/+d	*删除"main"下的行*
:.,$d x	*从这一行删除到最后一行，放入寄存器 x*

edit

e[!] [+*num*] [*filename*]

开始编辑名为*filename*的文件。如果未指定*filename*，则编辑当前文件的副本。加上!以编辑新文件，即使当前文件在前次改变后尚未保存。使用参数+*num* 时，则于第*num*行开始编辑。*num*可替换为模式或是/*pattern*的形式。

范例

:e file	*编辑位于当前缓冲区中的文件*

```
:e +/^Index #         编辑匹配模式的候补文件
:e!                   重新编辑当前文件
```

file

`f [`*filename*`]`

改变当前缓冲区的文件名为*filename*。下次缓冲区被写入时，它将被写入名为*filename*的文件。当名称改变时，缓冲区将被设置"not edited"标志，代表编辑的对象并非现有的文件。如果新文件名与磁盘上的现有文件相同，需使用:w!覆盖现有文件。指定文件名时，%用于表示当前的文件名。#能用于指示候补文件名。如果没有指定*filename*，则列出缓冲区的当前名称与状态。

范例

```
:f %.new
```

fold

address `fo`

折叠*address*指定的行。折叠（fold）把屏幕上的数行压缩成一行，稍后可再打开折叠。不会影响到文件里的文本。{Vim}

foldclose

`[`*address*`] foldc[!]`

关闭指定*address*中的折叠；若未指定，则关闭当前地址处的。加上!可关闭多层折叠。{Vim}

foldopen

`[`*address*`] foldo[!]`

打开指定*address*中的折叠；若未指定，则打开当前地址处的。加上!可打开多层折叠。{Vim}

global

[*address*] g[!]/*pattern*/[*commands*]

对包含*pattern*（模式）的所有行执行指定*commands*，或在指定*address*时于指定的范围内应用命令。如果未指定*commands*，则列出所有这样的行。加上!，对所有不包含*pattern*的所有行执行命令。请参考本节稍后列出的 v命令。

范列

`:g/Unix/p`	*列出所有包含"Unix"的行*
`:g/Name:/s/tom/Tom/`	*把包含"Name:"的每一行的tom换成Tom*

hide

hid

关闭当前的窗口，除非它是最后一个窗口，但不会从内存中移除缓冲区。对未保存的缓冲区使用本命令是安全的。{Vim}

insert

[*address*] i[!]
text
.

在指定的*address*前插入*text*；未指定时，则于当前的地址插入文本。加上！则在文本输入期间调换autoindent的设置。终止输入新文本的方式，是输入只包含点号的行。

join

[*address*] j[!] [*count*]

把指定范围内的文本合并成一行，具有空白调整功能，在点号（.）后提供两个空格，）前不可有空格，而其他情况用一个空格。加上!可避免空白的调整。

范例

`:1,5j!`	*合并最前面的5行，保留空白*

jumps

ju

列出使用CTRL-I与CTRL-I的跳转列表（jump list）。这份列表记录大多数跳过多行的移动命令。它在跳转前记录光标的位置。{Vim}

k

[*address*] k *char*

与mark相同，参见位于407页的mark命令。

left

[*address*] le [*count*]

向左对齐*address*指定的行；若未指定，则向左对齐当前的行。缩排空间由*count*决定。{Vim}

list

[*address*] l [*count*]

列出指定行，使tab字符呈现为^I，行尾字符则呈现为$。l就像临时版的:set list。

map

map[!] [*string commands*]

定义名为*string*的键盘宏为指定的*commands*序列。*string*通常是一个单一字符或#*num*序列，后者代表键盘上的功能键。使用!创建输入模式下的宏。没有参数的话，则列出当前定义的宏。

范例

:map K dwwP	*对调两个词*
:map q :w^M:n^M	*写入当前文件，转至下一个*
:map! + ^[bi(^[ea)	*把前一个词以括号括起*

注意： Vim有K与q命令，范例定义的别名将隐藏这两个命令。

mark

[*address*] ma *char*

以*char*（一个小写字母）标记特定行，与k命令相同。用'*x*即可回到该行（单引号加上*x*，*x*与*char*相同）。Vim中也可使用大写字母与数字字符做标记。小写字母的运作方式与vi相同。大写字母与文件名相关联，能用在多个文件间。然而，编号的标记是在特定的viminfo文件里被维护，且不能使用这个命令来设置。

marks

marks [*chars*]

列出*chars*指定的标记列表，若没有*chars*，则列出所有的当前标记。{Vim}

范例

`:marks abc`	*列出标记a、b、c*

mkexrc

mk[!]*file*

创建一个.exrc文件，包含改变ex选项与按键映射的set命令。本命令将保存当前的选项设置，我们可于稍后恢复设置。{Vim}

move

[*address*] m *destination*

移动*address*指定的行到*destination*地址。

范例

`:.,/Note/m /END/`	*移动文本块到包含“END”的行后*

new

[*count*] new

创建一个新窗口，高度为*count*行，带有空白缓冲区。{Vim}

next

n[!] [[+*num*] *filelist*]

编辑命令行参数列表中的下一个文件。使用args列出这些文件。如果提供了*filelist*，则把当前的参数列表替换成*filelist*，并开始编辑其中的第一个文件。加上+*num*参数时，编辑于第*num*行开始。*num*可替换为模式（形式为 /*pattern*）

范例

:n chap*	*开始编辑所有“chapter”文件*

nohlsearch

noh

使用hlsearch选项时，暂停高亮显示所有匹配搜索的内容。下次搜索时仍会继续出现。{Vim}

number

[*address*] nu [*count*]

列出*address*指定的每一行，前面附上它的缓冲区行编号。使用#作为number的替代缩写。*count*指定显示的行数，从*addres*指定的行开始计算。

only

on [!]

让当前的窗口成为屏幕上的唯一窗口。打开在调整缓冲区中的窗口不会从屏幕上被移除（改为隐藏），除非使用了!字符。{Vim}

open

[*address*] o [/*pattern*/]

在*address*指定的行或在匹配*pattern*的行进入开放模式（vi）。按Q离开。开放模式（open mode）让我们可以使用一般的vi命令，但一次只能应用一行。它对于慢速拨号网络可能很有用（或适用于距离非常遥远的国际互联网ssh连接）。

preserve

pre

保存当前的编辑器缓冲区中的内容，宛如系统即将崩溃一样。

previous

prev[!]

编辑命令行参数列表里的前一个文件。{Vim}

print

[*address*] p [*count*]

列出*address*指定的行。以*count*指定列出的行数，从*address*指定的地址开始。P是另一个缩写。

范例

:100;+5p	*显示第100行及其下5行*

put

[*address*] pu [*char*]

放置稍早从*char*指定的命名寄存器里删除或拖动的行的内容，放置位置为*address*指定的行。如果*char*未被指定，则恢复前次删除或拖动的文本。

qall

qa[!]

关闭所有窗口并终止当前的编辑会话。使用!丢弃自前次保存后的任何改变。{Vim}

quit

q[!]

终止当前的编辑会话。使用!丢弃自前次保存后的任何改变。如果编辑会话包括参数列

表中从未被访问过的额外文件，则可输入q!或输入两次q离开。Vim只在屏幕上还有其他窗口打开时关闭编辑窗口。

read

[*address*] r *filename*

在*address*指定的行后复制*filename*文件的文本。如果未指定*filename*，则使用当前的文件。

范例

```
:0r $HOME/data                    于当前文件的开始处读入文件
```

read

[*address*] r ! *command*

将shell *command*的输入读至*address*指定的行后的文本。

范例

```
:$r !spell %                      把拼写检查的结果放在文件结尾处
```

recover

rec [*file*]

从系统的保存区中恢复*file*。

redo

red

恢复前次未做的改变。与CTRL-R相同。{Vim}

resize

res [[±]*num*]

调整当前窗口的尺寸为*num*行高。如果指定了+或-，则当前的窗口增（或）减*num*行。{Vim}

rewind

`rew[!]`

倒回参数列表并开始编辑列表中的第一个文件。加上!，即使当前文件于前次改变后尚未保存，亦可倒回。

right

[*address*] ri [*width*]

把*address*指定的行向右对齐至第*width*栏；若未指定*address*，则把当前的行向右对齐。若未指定*width*，则采用textwidth选项。{Vim}

sbnext

[*count*] sbn [*count*]

分割当前的窗口并开始编辑后面的第*count*个缓冲区（在缓冲区列表中）。如果未指定*count*，则编辑缓冲区列表里的下一个缓冲区。{Vim}

sbuffer

[*num*] sb [*num*]

分割当前的窗口并开始于新窗口中编辑缓冲区列表里的第*num*个缓冲区。被编辑的缓冲区也能用文件名指定。若未指定缓冲区，则于新窗口中打开当前的缓冲区。{Vim}

set

se *parameter1 parameter2* ...

以每个*parameter*设置选项的值，或于未提供*parameter*时，列出所有改变过值而不再是默认值的选项。对布尔类型的选项，每个*parameter*可被解析为*option*或no*option*；其他选项的值则可通过语法*option=value*指派。指定all以列出当前的设置。set *option*?的形式显示*option*的值。请参考附录B中列出set选项的表格。

范例

```
:set nows wm=10
:set all
```

shell

sh

创建一个新shell。于shell终止后恢复编辑。

snext

[*count*] sn [[+*num*] *filelist*]

分割当前的窗口并开始编辑命令行参数列表里的下一个文件。如果提供了*count*，则编辑往后数的第*count*个文件。如果提供了*filelist*，则把当前的参数列表替换为*filelist*并开始编辑其中的第一个文件。当有+*n*参数时，则编辑开始于第*num*行。*num*可替换为模式（形式为/*pattern*）。{Vim}

source

so *file*

从*file*读入（来源）并执行ex命令。

范例

```
:so $HOME/.exrc
```

split

[*count*] sp [+*num*] [*filename*]

分割当前的窗口并于新窗口中载入名为*filename*的文件；若未指定文件，则两个窗口中都载入相同的缓冲区。让新窗口的高度为*count*行；若未指定*count*，则将窗口平均分割为两个。使用+*num*参数时，将于第*num*行开始编辑。*num*可替换为模式（形式为/*pattern*）。{Vim}

sprevious

[*count*] spr [+*num*]

分割当前的窗口并于新窗口中开始编辑命令行参数列表中的前一个文件。如果指定了*count*，则编辑往前数的第*count*个文件。使用+*num*参数时，于第*num*行开始编辑。*num*可替换为模式（形式为/*pattern*）。{Vim}

stop

st

暂停编辑会话。与CTRL-Z相同。使用shell的fg命令以继续会话。

substitute

[*address*] s [/*pattern*/*replacement*/] [*options*] [*count*]

在每个指定的行上以*replacement*取代*pattern*的第一个实例。如果省略了*pattern*与*replacement*，则重复前次的替换。*count*指定替换执行的行数，从*address*的位置开始（在Solaris vi中，本命令的全名不能运作）。

选项

c 在每次改变前出现确认改变的提示。

g 替换每一行（全局）上*pattern*的所有实例。

p 列出替换执行的最后一行。

范例

```
:1,10s/yes/no/g                替换最初的10行
:%s/[Hh]ello/Hi/gc             确认全局替换
:s/Fortran/\U&/ 3              把接下来3行的"Fortran"改为大写的
:g/^[0-9][0-9]*/s//Line &:/    开头处具有一或多个数字的每一行均加上"Line"与一个冒号
```

suspend

su

暂停编辑会话。与CTRL-Z相同。使用shell的fg命令以继续会话。

sview

[*count*] sv [+*num*] [*filename*]

与split命令相同，但为新缓冲区设置readonly选项。{Vim}

t

[*address*] t *destination*

复制*address*中包含的行到（to）指定的*destination*地址。t等效于copy。

范例

```
:%t$
```
复制文件并将其添加到末尾

tag

[*address*] ta *tag*

在**tags**文件中，寻找匹配*tag*的文件与行并由此开始编辑。

范例

运行ctags，然后切换到包含*myfunction*的文件：

```
 :!ctags *.c
:tag myfunction
```

tags

tags

列出标签栈中的标签列表。{Vim}

unabbreviate

una *word*

从缩写列表中移除*word*。

undo

u

逆转（或称撤销）前次编辑命令所做的改变。在vi中，连续两次使用该命令将自我撤销，重做刚被撤销的改变。Vim支持多次撤销命令。在Vim中可使用redo重做刚被撤销的改变。

unhide

[*count*] unh

分割屏幕，使缓冲区列表里每个活跃的缓冲区都有自己的显示窗口。如果指定了，则限制窗口数为*count*。{Vim}

unmap

unm[!] *string*

从键盘宏列表移除*string*。使用!以移除插入模式的宏。

v

[*address*] v/*pattern*/[*command*]

对包含*pattern*的所有行执行*command*。如果未指定*command*，则列出所有这样的行。v等效于g!。请参考稍早出现在本节中的global命令。

范列

:v/#include/d	*删除所有行，包含"#include"的行除外*

version

ve

列出编辑器的当前版本编号及前次改变的日期。

view

vie[[+*num*] *filename*]

与edit相同，但设置文件为readonly。在ex模式中执行时，回到正常或可视模式。{Vim}

visual

[*address*] vi [*type*] [*count*]

在*address*指定的行进入可视模式（vi）。按Q回到ex模式。*type*可为-、^、.的其中之一（请参考本节稍后的z命令）。*count*指定初始的窗口尺寸。

visual

vi [+*num*] *file*

开始在视觉模式（vi）中编辑*file*，可选择从第*num*行开始。*num*可替换为模式（采用/*pattern* 形式）。{Vim}

vsplit

[*count*] vs [+*num*] [*filename*]

与split命令相同，但垂直分割屏幕。*count*参数用于指定新窗口的宽度。{Vim}

wall

wa[!]

以文件名写入所有已改变的缓冲区。加上!以强迫写入任何标记为readonly的缓冲区。{Vim}

wnext

[*count*] wn[!] [[+*num*] *filename*]

写入当前的缓冲区并打开参数列表中的下一个文件；若指定*count*，则打开后面的第*count* 个文件。如果指定了*filename*，则编辑该文件。使用+*num*参数时，从第*num*开始编辑。*num*可替换为模式（采用/*pattern*形式）。{Vim}

wq

wq[!]

写入并离开文件，一气呵成。文件总会被写入。!标志强迫编辑器写入*file*的任何当前内容。

wqall

wqa[!]

写入所有已改变的缓冲区并离开编辑器。加上!以强迫写入任何标记为readonly的缓冲区。xall是本命令的另一个别名。{Vim}

write

[*address*] w[!] [[>>]*file*]

写入*address*指定的行至*file*；若未指定*address*，则缓冲区中的全部内容都写入，如果连*file*都省略，则把缓冲区内容存入当前的文件中。如果使用>>*file*，则把行内容附加到指定*file*的结尾。加上!以强迫编辑器覆盖*file*当前的任何内容。

范例

```
:1,10w name_list          复制前10行到文件name_list
:50w >> name_list         现在附加第50行
```

write

[*address*] w ! *command*

写入*address*指定的行到*command*中。

范例

```
:1,66w !pr -h myfile | lp          列出文件的第一页
```

X

X

提示加密密钥。:set key更好，在输入密钥时不会回显到控制台上。需移除加密密钥时，只需把key选项重设为空白值。{Vim}

xit

x

如果文件自上次写入后有改变，即写入文件，然后离开。

yank

[*address*] y [*char*] [*count*]

把*address*指定的行放入指名的寄存器*char*。寄存器名称为小写字母a到z。大写字母表示把文本附加到相对应的寄存器里。如果没有指定*char*，则把内容放入通用寄存器。*count*指定拖动的行数，从*address*指定的位置开始。

范例

```
:101,200 ya a                复制第100~200行至寄存器“a”
```

---

z

[*address*] z [*type*] [*count*]

列出一个文本窗口，其中由*address*指定的行位于窗口顶端。*count*指定显示的行数。

类型

+
: 指定行放置于窗口顶端（默认值）。

-
: 指定行放置于窗口底端。

.
: 指定行放置于窗口中间。

^
: 列出前一个窗口。

=
: 指定行放置于窗口中间并留置当前的行于这一行。

&

[*address*] & [*options*] [*count*]

重复前面的替换（s）命令。*count*指定欲替换的行数，从*address*指定的位置开始。*option*与替换命令的选项相同。

范例

```
:s/Overdue/Paid/             在当前的行上做一次替换
:g/Status/&                  在所有“Status”行上重做替换
```

@

[*address*] @ [*char*]

执行*char*指定的寄存器的内容。如果给定了*address*，则先把光标移到指定的地址。如果*char*是@，即重复前一次的@命令。

=

[*address*] =

列出*address*所指文本行的行号。默认为最后一行的行号。

!

[*address*] ! *command*

在shell中执行Unix *command*。如果指定了*address*，则使用包含*address*的行作为*command*的标准输入，并以输出和错误输出取代这些行（这称为通过命令[*command*]过滤[*filtering*]文本）。

范例

```
:!ls                        列出当前目录中的文件
:11,20!sort -f              排列当前文件中第11~第20行的顺序
```

<>

[*address*] < [*count*]
  or
[*address*] > [*count*]

向左（<）或向右（>）移动*address*指定的行。在移动行时，只会在行的开头处加上（或删除）空格或tab。*count*可指定欲移动的行数，由*address*指定行开始计算。shiftwidth选项控制移动的栏数。重复<或>，可递增移动量。例如，:>>>移动的距离即为:>的三倍。

~

[*address*] ~ [*count*]

以最近一次的s（替换）命令使用的替换模式，替换最后被使用的正则表达式（即使正则表达式来自搜索而非s命令）。这项命令不太常用，请参考第六章以了解细节。

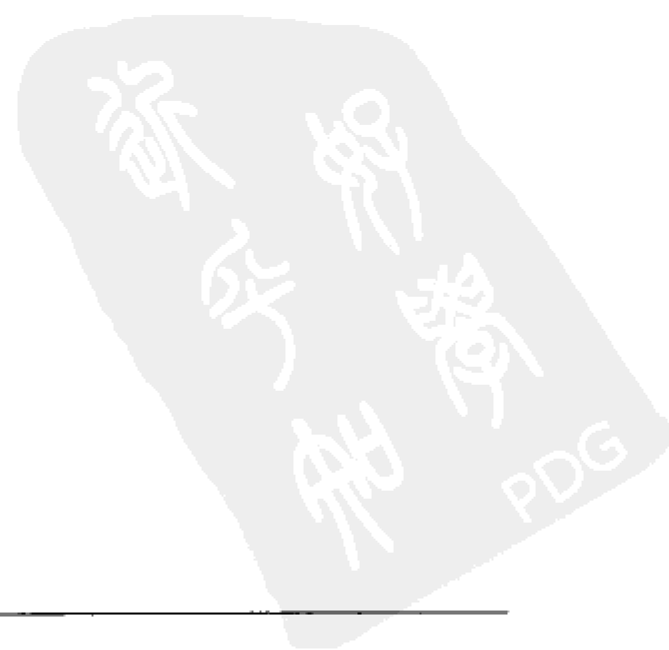
address

address

列出*address*中指定的行。

ENTER

列出文件中的下一行（只限用于ex，不能从vi中的:提示符使用）。



附录B

设置选项

本附录介绍Solaris vi、nvi 1.79、elvis 2.2、Vim 7.1与vile 9.6中重要的set命令选项。

Solaris的vi选项

表B-1包含了重要的set命令选项的简短描述。在第一栏中，选项是按字母顺序排列的，如果选项可以缩写，则列在括号中。第二栏显示vi的默认值，除非明确使用 set 命令改变设置（可以手动设置或在.exrc文件中加入）。最后一栏描述了启用选项时的选项行为。

表B-1：Solaris vi的设置选项

选项	默认值	说明
autoindent (ai)	noai	在插入模式中，对每一行做与上一行或下一行相同的缩排。与shiftwidth选项一起使用
autoprint (ap)	ap	在每一个编辑器命令执行之后显示变动的部分（若是全局替换，则显示最后一处替换）。
autowrite (aw)	noaw	如果在使用:n打开另一个文件或用:!执行Unix命令前当前的文件已经改变，则自动写入（保存）文件
beautify (bf)	nobf	忽略所有输入时的控制字符（除了tab、换行与换页符）
directory (dir)	/tmp	命名ex/vi存储缓冲区文件的目录（此目录必须允许写入）

表B-1：Solaris vi的设置选项（续）

选项	默认值	说明
edcompatible	noedcompatible	记住最近一次替换命令所用的标志（全局，确认），并将其用在下一个替换命令上。尽管名义上如此，但是并没有哪个版本的ed是实际这样做的
errorbells (eb)	errorbells	发生错误时发出响铃
exrc (ex)	noexrc	允许执行不在用户主目录中的.exrc文件
flash (fp)	nofp	以闪烁屏幕取代发出响铃
hardtabs (ht)	8	定义终端机硬件定位的边界
ignorecase (ic)	noic	搜索时不考虑大小写
lisp	nolisp	插入缩排时使用适当的Lisp格式。()、{}、[[、]]要被修改，才能在Lisp中具有意义
list	nolist	将tab字符打印为^I；行尾则标记为$（使用list来告知结束字符是tab还是空格）
magic	magic	通配符.（点号）、*（星号）与[]方括号）在模式中有特别意义
mesg	mesg	在vi中编辑时，允许系统消息显示在终端上
novice	nonovice	要求使用长的ex命令名称，像copy或read
number (nu)	nonu	在编辑会话期间于屏幕左方显示行编号
open	open	允许从ex进入开放（open）模式或可视（visual）模式。虽然Solaris vi中没有这些模式，但这个选项传统上会保留在vi中，你的Unix版本的vi中有可能也有
optimize (opt)	noopt	在打印多行文本时，取消行尾的换行字符（carriage return）；在简易终端（dumb terminal）上打印开头有空白（空格与tab字符）的行时，如此可增加速度

表B-1：Solaris vi的设置选项（续）

选项	默认值	说明
paragraphs (para)	IPLPPPQP LIpplpipbp	定义用{和}移动时使用的段落定界符。此值中的一对字符是开始段落的troff宏名称
prompt	prompt	使用vi的Q命令时显示ex的提示符号（:）
readonly (ro)	noro	任何写入（保存）文件的动作均失效，除非写入时加上！（可与w、ZZ、autowrite一起使用）
redraw(re)		vi会在任何编辑动作发生时重绘屏幕（换句话说，插入模式会挤入现有字符，而删除的行会立刻消失）。默认值依速度与终端类型而不同。noredraw在速度较慢的简易终端上比较有用：删除的行会显示成@，而插入的文本看起来会覆盖现有的文本，除非按下ESC
remap	remap	允许嵌套的映射序列
report	5	当编辑动作影响某个数量以上的行时，在状态行中显示消息。例如，6dd的回报消息是“6 lines deleted”
scroll	*[半个窗口]*	使用^D与^U命令时滚动的行数
sections (sect)	SHNHH HU	定义[[与]]移动时的小节定界符。此值中的一对字符是开始小节的troff宏名称
shell (sh)	/bin/sh	用作shell转义（:!）与shell命令（:sh）的shell路径名称。默认值从 shell环境中取得，在不同的系统上会有不同的值
shiftwidth (sw)	8	定义使用autoindent选项时，往回（^D）的tab字符所使用的空格数目也用于<<与>>命令
showmatch (sm)	nosm	在vi中，当输入)或}时，光标会短暂移动到对应的(或{（如果找不到，则会发出错误消息的响铃）。在编程时很有用

表B-1：Solaris vi的设置选项（续）

选项	默认值	说明
showmode	noshowmode	在插入模式中，于提示行显示一个消息，表示当前的插入类型。例如“OPEN MODE”或“APPEND MODE”
slowopen (slow)		在插入时暂缓显示。默认值会依行速与终端类型而不同
tabstop (ts)	8	定义编辑会话期间tab字符缩排的空格数目（打印机仍然会使用系统定义的tab字符，其值为8）
taglength (tl)	0	定义标签中有效字符的数目。默认值（0）表示所有的字符都是有效的
tags	tags /usr/lib/tags	定义包含标签的文件的路径名称列表（参阅Unix的ctags命令）。默认情况下，vi会搜索当前目录与/usr/lib/tags中的tags文件
tagstack	tagstack	在栈上启用标签堆栈
term		设置终端类型
terse	noterse	显示较短的错误消息
timeout (to)	timeout	键盘映射会在一秒钟后失效*
ttytype		设置终端类型。这只是term的另一个名称
warn	warn	显示警告消息“No write since last change”
window (w)		在屏幕上显示文件中一定数量的行。默认值会依行速与终端类型而不同
wrapmargin (wm)	0	定义右边界。如果大于0，则会自动插入换行符以换行
wrapscan (ws)	ws	搜索会在遇到文件两端时自动绕回
writeany (wa)	nowa	允许保存到任何文件中

* 在映射了几个键后（例如:map zzz 3dw），你可能想改用notimeout；否则，必须在一秒内输入 zzz。当你在插入模式里映射键时（例如:map! ^[OB ^[ja），应使用timeout；否则，vi 在我们键入其他键前不能对ESC作出反应。

nvi 1.79的选项

nvi 1.79一共有78个选项会影响其行为。表B-2汇总了其中最重要的一些。在表B-1中介绍过的选项不再于这里重复。

表B-2：nvi 1.79的设置选项

选项	默认值	说明
backup		一个描述要使用的备份文件名的字符串。当前的文件内容在存储成新的数据前会先存在这个文件中。首字符为N，可让nvi于文件名后加上编号，版本编号永远逐渐增加。"N%.bak" 就是个不错的范例
cdpath	CDPATH环境变量，或是当前目录	:cd命令的搜索路径
cedit		在冒号命令行输入这个字符串的第一个字符时，nvi会打开一个命令历史记录的新窗口，我们可以编辑该窗口。在其中任何一行上按下ESC，会执行此行的内容。ENTER是这个选项的一个好选择（使用^V ^[来输入）
comment	nocomment	如果第一个非空行以 /*、// 或 # 开始，则nvi先跳过注解文本再显示文件。如此可避免显示冗长、无聊的法律等注意事项
directory (dir)	TMPDIR环境变量，或是/tmp	nvi放置其临时文件的目录
extended	noextended	用egrep风格的扩展正则表达式来进行搜索
filec		在冒号命令行输入这个字符串的第一个字符时，nvi会将光标前以空格分隔的词视为词后附加*，并做shell样式的文件名扩展。ESC是这个选项的一个好选择（使用^V ^[来输入）。当这个字符与cedit选项相同时，只有当这个字符是输入冒号命令行中的第一个字符时，才会执行命令行的编辑动作

表B-2：nvi 1.79的设置选项（续）

选项	默认值	说明
iclower	noiclower	让所有的正则表达式搜索都不考虑大小写，只要搜索模式中不包含大写字母
leftright	noleftright	过长的行由屏幕左端往右端滚动，而不会绕回
lock	lock	nvi会尝试对文件取得独占锁。编辑一个不能被锁定的文件将创建只读会话
octal	nooctal	未知的字符以八进制显示，而不用十六进制
path		一个以冒号分隔的目录列表，nvi会从中查找要编辑的文件
recdir	/var/tmp/vi.recover	存储恢复文件的目录
ruler	noruler	显示光标所在的行与列
searchincr	nosearchincr	使用增量搜索
secure	nosecure	关闭通过文本过滤（:r!、:w!）对外部程序的访问，禁用vi模式的!与^Z命令以及ex模式的!、shell、stop与suspend命令。一旦设置之后，就不能再改变
shellmeta	~{[*?$`'"\	当这些字符中任何一个出现在ex命令的文件名参数中时，此参数会被shell选项中指名的程序展开
showmode (smd)	noshowmode	在状态行显示一个字符串，指示当前的模式。如果文件被修改过，则显示*
sidescroll	16	当leftright为true时，显示屏幕向左或向右移动的列数
taglength (tl)	0	定义标签中有效字符的数目。默认值（0）表示所有的字符都是有效的
tags (tag)	tags /var/db/libc.tags /sys/kern/tags	可能的标签文件的列表
tildeop	notildeop	~命令采取相关的移动，而不只是使用前缀的计数值

表B-2：nvi 1.79的设置选项（续）

选项	默认值	说明
wraplen (wl)	0	与wrapmargin选项相同，但本选项指示从左边界开始经过多少个字符才会被分割。wrapmargin的值会覆盖wraplen

elvis 2.2的选项

elvis 2.2一共有225个选项会影响其行为。表B-3汇总了其中最重要的一些。大部分在表B-1中介绍过的选项不再于这里重复。

表B-3：elvis 2.2的设置选项

选项	默认值	说明
autoiconify (aic)	noautoiconify	在新窗口离开图标状态时，将旧窗口缩成图标状态。只限X11
backup (bk)	nobackup	在把当前的文件写入磁盘前先产生备份文件（xxx.bak）
binary (bin)		缓冲区中的数据不是文本。这个选项由系统自动设置
boldfont (xfb)		名称采用黑体字。只限X11
bufdisplay (bd)	normal	缓冲区默认的显示模式（hex、html、man、normal、syntax、tex）
ccprg (cp)	cc ($1?$1:$2)	:cc对应的shell命令
directory (dir)		存储临时文件的地方。默认值依系统而不同
display (mode)	normal	当前显示模式的名称，由:display命令设置
elvispath (epath)		可供搜索配置文件的目录列表。默认值依系统而不同
focusnew (fn)	focusnew	强制将键盘的焦点移到新的窗口。只限X11
font(fnt)		名称采用正常字体。用于Windows与X11界面
gdefault (gd)	nogdefault	让替换命令更改所有实例

表B-3：elvis 2.2的设置选项（续）

选项	默认值	说明
home (home)	$HOME	文件名中的～代表的主目录
italicfont (xfi)		斜体字体的名称。只限X11
locked(lock)	nolocked	使缓冲区为只读的，并使大多数修改缓冲区的命令失败。对只读的HTML文件，通常系统会自动设置
lpcolor(lpcl)	nolpcl	打印时使用彩色，用于:lpr
lpcolumns (lpcols)	80	打印机的页宽度，用于:lpr
lpcrlf (lpc)	nolpcrlf	打印机需要用CR/LF表示文件中的换行，用于:lpr
lpformfeed	nolpformfeed	在每一页的最后插入换页符号，用于:lpr
lpheader(lph)	nolph	在每一页的顶端列出标题，用于:lpr
lplines (lprows)	60	打印机的页长度，用于:lpr
lpout (lpo)		:lpr所用的打印机文件或过滤器。典型的值应该是:lpr。默认值依系统而不同
lptype (lpt)	dumb	打印机类型，用于:lpr。其值可能是ps、ps2、epson、pana、ibm、hp、cr、bs、dumb、html、ansi的其中之一
lpwrap (lpw)	lpwrap	模仿行绕回，用于:lpr
makeprg (mp)	make $1	:make所使用的shell命令
prefersyntax(psyn)	never	控制语法模式的使用。对于HTML和手册页（manpages）呈现输入（取代格式化内容）很有用。加上writable选项时，应用至可写入文件。加上local选项时，应用至当前目录中的文件
ruler (ru)	noruler	显示光标所在的行与列

表B-3：elvis 2.2的设置选项（续）

选项	默认值	说明
security(sec)	normal	设置为normal（标准vi行为）、safer（试图阻止写入恶意脚本）、restricted（试图以受限编辑器的使用方式，维护 elvis的安全）的其中之一。通常应使用:safely命令设置系统的安全级别，不应直接设置
showmarkups (smu)	noshowmarkups	在man与html模式中，显示光标位置处的标记，在其他模式中则不显示
sidescroll (ss)	0	往左、右的移动量。0会模仿vi行为，使行绕回
smartargs(sa)	nosmartargs	在输入函数名称与function字符后（通常是左括号），把查找tags文件而找到的函数参数放到屏幕上
spell(sp)	nospell	对拼写错误的地方高亮显示。也能应用在脚本里，需根据查询tags文件的结果
taglength (tl)	0	定义标签中有效字符的数量。默认值（0）表示所有的字符都是有效的
tags (tagpath)	tags	可能的标签文件的列表
tagstack (tsk)	tagstack	记住栈上的标签搜索来源
undolevels (ul)	0	可撤销的命令数目。0表示模仿vi。你可能想要将此值设置得较大
warpback (wb)	nowarpback	在离开时，将光标移回启动 elvis 的 xterm。只限X11
wrapto (wt)	don't	^W ^W影响光标移动的方式：don't表示不移动，scrollbar会将指针移到滚动条上，origin将指针移到左上角，而corners将指针移到离当前光标位置最远与最近的角上。这个选项让X显示画面能够移动，确保窗口全部都在屏幕内

Vim 7.1 选项

Vim 7.1一共有295个选项会影响其行为。表B-4汇总了其中最重要的一些。大部分在表B-1 中介绍过的选项不再于这里重复。

表中的解释比较简洁扼要。大部分的选项信息都可以在Vim的在线帮助中取得。

表B-4：vim 7.1 的设置选项

选项	默认值	说明
autoread(ar)	noautoread	检测Vim里的文件是否被外部修改过，而不是被Vim修改，并以改变过的文件版本自动刷新Vim的缓冲区
background (bg)	dark或light	Vim会尝试使用适合特定终端的背景与前景颜色。默认值根据当前的终端或窗口系统而不同
backspace (bs)	0	控制是否可使用退格键经过换行处与插入动作开始处。其值包括：0 代表与vi兼容性；1表示可经过换行处；2 表示可经过插入操作开始处；3表示两者皆可
backup (bk)	nobackup	在覆盖文件之前先做备份，接着在文件成功写入之后仍然恢复原状。如果只要在文件写入时产生备份文件，则使用writebackup选项
backupdir (bdir)	., ~/tmp/, ~/	存储备份文件的目录列表，以逗号隔开。备份文件创建在列表中的第一个可用目录中。如果此值为空，则不能创建备份文件。名称.（点号）表示其与编辑文件所在的目录相同
backupext (bex)	~	附加在文件名中形成备份文件名的字符串
binary (bin)	nobinary	改变一些其他选项，以便编辑二进制文件。这些选项稍早采用的值会被记住，离开bin状态时会被恢复。每一个缓冲区都有自己的已保存选项值。这个选项应该在编辑二进制文件前设置。你也可以使用命令行选项-b
cindent (cin)	nocindent	启用自动的C程序智慧缩排

表B-4：vim 7.1 的设置选项（续）

选项	默认值	说明
cinkeys (cink)	0{,0},:,0#,!^F, o,O,e	一些按键的列表，于插入模式中输入时，使当前这行重新缩排。只在cindent打开时有效
cinoptions (cino)		影响cindent重新缩排C程序的方式。详细内容请参阅在线帮助
cinwords (cinw)	if, else, while, do, for, switch	如果smartindent或cindent被设置时，这些关键字会在下一行开启一个新的缩排方式。对于cindent，这只会在适合的地方发生（在{...}之内）
comments (com)		一个以逗号分隔的字符串列表，可以开启一行注释。详细内容请参阅在线帮助
compatible (cp)	cp；发现.vimrc文件时则为nocp	让Vim在许多方面与vi更类似，多到在此不能细述。默认是打开的，以避免令人吃惊的结果。如果拥有.vimrc文件，会关闭vi兼容性，这通常是我们想要的副作用
completeopt (cot)	menu, previes	用于插入模式自动补全的选项列表（以逗号分隔选项）
cpoptions (cpo)	aABcdFs	一个单一字符标志的序列，各个标志分别表示一种Vim是否会正确模仿vi的方式。当此值为空时，使用Vim的默认值。详细内容请参阅在线帮助
cursorcolumn (cuc)	nocursorcolumn	以CursorColumn高亮显示屏幕上光标所在的栏。对于垂直排列文本很有用。有可能减慢屏幕的显示
cursorline (cul)	nocursorline	以CursorRow高亮显示屏幕上光标所在的栏。使较容易在编辑会话中找到当前的行。与cursorcolumn一起使用，可得到准确定位光标的效果。有可能减慢屏幕的显示

表B-4：vim 7.1 的设置选项（续）

选项	默认值	说明
define (def)	^#\s*define	一个描述宏定义的搜索模式。默认值是针对 C 程序的。对C++来说，可以使用^\(#\s*define \ |[a-z]*\s*const\s*[a-z]*\)。当使用:set命令时，需用两个反斜线
directory (dir)	., ~/tmp, /tmp	交换区文件所在的目录名称列表，以逗号隔开。交换区文件会被创建在第一个可用目录。如果此值为空，则不会使用交换区文件，而且不能进行恢复（recovery）！名称.（点号）表示将交换区文件放在与编辑文件所在相同的目录。推荐将.放在列表中的第一位，如果编辑同一个文件两次，将产生警告
equalprg (ep)		用于=命令的外部程序。当此选项为空时，会使用内部的格式化函数
errorfile (ef)	errors.err	快速修复模式所用的错误文件名。在命令行中使用-q参数时，errorfile即设置为后接参数
errorformat (efm)	（过长的文件在此不打印出）	与scanf类似的格式描述，用于错误文件中的行
expandtab (et)	noexpandtab	在插入tab符号时，将其展开为适当数量的空格
fileformat (ff)	Unix	描述在读入或写入当前的缓冲区时，表示一行结束的方式。可能的值为dos（CR-LF）、Unix（LF）与mac（CR）。通常Vim 会自动设置此值
fileformats (ffs)	dos,Unix	列出Vim在读入文件时会尝试的一行结束方式。如果有多个名称，则在读入文件时自动检测一行的结束
formatoptions (fo)	Vim默认值：tcq，vi默认值：vt	描述如何自动格式化的字母序列。详细内容请参阅在线帮助
gdefault (gd)	nogdefault	让替换命令更改所有实例
guifont (gfn) 试		在启动图形用户界面版本的Vim时，尝使用此处的字体列表（以逗号分隔）

表B-4：vim 7.1 的设置选项（续）

选项	默认值	说明
hidden (hid)	nohidden	如果当前缓冲区从窗口中卸载，则隐藏此缓冲区，而不是将其舍弃
history (hi)	Vim默认值：20；vi默认值：0	控制命令历史记录中保存的ex命令、搜索字符串与表达式的数量
hlsearch (hls)	nohlsearch	特别标置最近一次搜索模式的所有匹配之处
icon	noicon	Vim尝试改变所在的窗口所关联的图标名称。会被iconstring选项覆盖
iconstring		用于窗口的图标名称的字符串值
include (inc)	^#\s*include	定义查找include命令的搜索模式。默认值是C程序所用的
incsearch (is)	noincsearch	启用增量搜索
isfname (isf)	@,48-57,/,.,-,_ , +,,,$,:,~	可以包含在文件与路径名称中的字符列表。非Unix系统会有不同的默认值。@字符表示任何字母字符。它也会用在以下的其他is*XXX*选项，请见后续示例
isident (isi)	@,48-57,_,192-255	可以包含在标识符中的字符列表。非Unix系统会有不同的默认值
iskeyword (isk)	@,48-57,_,192-255	可以包含在关键字中的字符列表。非Unix系统会有不同的默认值。关键字用于搜索且可被许多命令认别，如w、[i等
isprint (isp)	@,161-255	列出可以直接显示在屏幕上的字符列表。非Unix系会有不同的默认值
makeef (mef)	/tmp/vim##.err	:make命令使用的错误文件名。非Unix系统会有不同的默认值。##将被数值替换，以形成唯一的名称
makeprg (mp)	make	:make命令所使用的程序。其值中的%与#会被展开
modifiable (ma)	modifiable	关闭此选项时，不允许对缓冲区的任何写入

表B-4：vim 7.1 的设置选项（续）

选项	默认值	说明
mouse		在非图形用户界面版本的 Vim 中启用鼠标。可用在MS-DOS、Win32、QNX pterm、xterm中。详细内容请参阅帮助文件
mousehide (mh)	nomousehide	在输入时隐藏鼠标指针。移动鼠标时恢复指针
paste	nopaste	更改许多选项，使得在Vim窗口中用鼠标做粘贴动作时，不会破坏粘贴的文本。如果将其关闭，会将这些选项恢复成原值。详细内容请参阅在线帮助
ruler (ru)	noruler	显示光标位置所在的行号与列号
secure	nosecure	在启动文件中禁用某些种类的命令。如果你使用的不是.vimrc与.exrc文件，则会自动启用
shellpipe (sp)		将:make的输出捕获到文件中所用的shell字符串。默认值依shell而有不同
shellredir (srr)		将过滤器输出捕获到临时文件中所用的shell字符串。默认值依shell而有不同
showmode (smd)	Vim默认值：smd， Vi默认值：nosmd	在状态行中放置插入、替代与可视模式的消息
sidescroll (ss)	0	水平方向滚动的栏数。值为 0 时表示将光标放在屏幕的正中央
smartcase (scs)	nosmartcase	如果搜索模式中包含大写字母，则会覆盖ignorecase选项
spell	nospell	打开拼写检查
suffixes	*.bak,~,.o,.h, .info,.swp	当进行文件名自动补全时，如果有多个文件匹配模式，此变量值将设定其先后顺序，以便选择Vim真正会使用的文件
taglength (tl)	0	定义标签中有效的字符数。默认值（0）表示所有的字符都有效

表B-4：vim 7.1 的设置选项（续）

选项	默认值	说明
tagrelative (tr)	Vim默认值：tr， vi默认值：notr	tags文件中来自其他目录的文件名，会以相对于tags文件所处的目录而计算其位置
tags (tag)	./tags,tags	:tag命令所用的文件名，以冒号或逗号分隔。开头的./会被当前文件的完全路径所替换
tildeop (top)	notildeop	让~命令的行为与运算符一样
undolevels (ul)	1000	可以撤销改变的最大次数。0值表示与vi兼容——撤销一层后，u撤销自己的动作。非Unix的系统可能会有不同的默认值
viminfo (vi)		在启动时读入viminfo文件，并在离开时写入。其值很复杂，它控制了Vim会存储到文件中的不同种类信息。详细内容请参阅在线帮助
writebackup (wb)	writebackup	在覆盖文件之前创建备份文件。在成功写入之后，备份会被删除，除非backup选项也被打开

vile 9.6选项

vile 9.6 一共有167个选项（在vile中称为“模式”），分别根据其用途，注记为*universal*、*buffer*或*window*模式。还有101个环境变量（environment variable），它们在脚本中的作用，大于用于直接的用户操作（注1）。并非每个平台上都能使用所有选项，有些只能应用在X11或Win32上。

表B-5汇总了vile最重要选项的编译时默认值。初始脚本，例如vileinit.rc，可能覆盖一些本表所列的值。大部分在表B-1中介绍过的选项不再于此重复。

注1： 包括被设置或用作其他命令的副作用的变量。由于它们的重点在于脚本，所以不太适合用于本表说明其作用——多半需要很长的篇幅，有兴趣的话，请参考在线帮助。

表B-5：vile 9.6的设置选项

选项	默认值	说明
alt-tabpos	noatp	控制光标应位于代表tab字符的空白的左方或右方
animated	animated	在草稿（scratch）缓冲区的内容在别处改变时自动更新其内容
autobuffer (ab)	autobuffer	使用“最近一次使用的”缓冲方式：依使用的顺序对缓冲区排序。否则，缓冲区维持开始编辑时的顺序
autocolor (ac)	0	自动根据语法着色。如果设置为零，则禁用自动语法着色。否则，应该设置为较小的正整数，代表调用autocolor-hook前需等待“quiet interval”的毫秒数
autosave (as)	noautosave	自动保存文件。在插入文本的每个autosavecnt字符后写入文件
autosavecnt (ascnt)	256	指定在插入多少个字符后便进行自动保存
backspacelimit (bl)	backspacelimit	如果禁用此选项，在插入模式中后退即可越过插入模式开始的位置
backup-style	off	控制在写入文件时要如何创建备份文件。可能的值包括off，表示不备份；.bak，表示DOS风格的备份；tilde，表示Unix下Emacs风格的hello.c~备份
bcolor	default	如果系统支持，便设置背景颜色
byteorder-mark	auto (bom)	控制用于分辨不同UTF编码类型的前缀检查。默认值auto要求vile检查文件；指定值则让vile使用该值
check-modtime	nocheck-modtime	如果文件在上一次读入或写入后被更改过，便发出“file newer than buffer”的警告消息，并提示要求确认
cindent	nocindent	启用C风格的缩排格式。这有助于在插入内容时自动维持当前的缩排层次，类似autoindent

表B-5：vile 9.6的设置选项（续）

<table>
<tr><th>选项</th><th>默认值</th><th>说明</th></tr>
<tr><td>cindent-chars</td><td>:#{}()[]</td><td>由cindent模式解释的字符列表。包括表示缩排到第1栏的#；表示多一层缩排的:，像在标签（label）后一样。列出的字符对也在fence-pairs选项里，使得被这些字符对括起的文本多一层缩排</td></tr>
<tr><td>cmode</td><td>off</td><td>内置的C代码的主模式</td></tr>
<tr><td>color-scheme (cs)</td><td>default</td><td>具体指定一个fcolor、bcolor、video-attrs、$palette的命名集合（由define-color-scheme命令定义）</td></tr>
<tr><td>comment-prefix</td><td>^\s*\(\(\s*[#*>]\)\|\(///*\)\)\+</td><td>描述在重新格式化注释时必须留下的一行的开头部分。默认值适用于Makefile、shell与C的注释以及电子邮件</td></tr>
<tr><td>comment</td><td>^\s*/\?\(\s*[#*>] \)\+/\?\s*$</td><td>定义注释的段落定界符所使用的正则运算式。其目的是在重新格式化时保留注释中的段落</td></tr>
<tr><td>cursor-tokens</td><td>regex</td><td>控制vile是否使用正则表达式或字符类，为各种命令解析屏幕上的标记（token）。可使用的值为：both、cclass、regex</td></tr>
<tr><td>dirc</td><td>nodirc</td><td>vile会在扫描目录以补全文件名时检查每一个名称。这可让你在提示符中分辨目录名称与文件名</td></tr>
<tr><td>dos</td><td>nodos</td><td>在读入文件时将CR-LF中的CR去除，并在写入文件时将其回放。对于不存在文件的新缓冲区，则会继承操作系统的行样式，而不管dos的值</td></tr>
<tr><td>fcolor</td><td></td><td>如果系统支持，便设置前景颜色</td></tr>
<tr><td>fence-begin
fence-end</td><td>/*
*/</td><td>表达简单的不可嵌套的围栏（fence）的开始与结束的正则表达式，例如C的注释</td></tr>
<tr><td>fence-if
fence-elif
fence-else
fence-fi</td><td>^\s*#\s*if
^\s*#\s*elif\>
^\s*#\s*else\>
^\s*#\s*endif\></td><td>标记面向行的嵌套围栏的开始、“else if”、“else”与结束的正则表达式，如C的预处理器控制行</td></tr>
<tr><td>fence-pairs</td><td>{}()[]</td><td>每一对字符表示一组应该以%映射的“围栏”</td></tr>
</table>

表B-5：vile 9.6的设置选项（续）

选项	默认值	说明
file-encoding	auto	指定缓冲区内容的字符编码，例如8bit、ascii、auto、utf-8、utf-16或utf-32的其中之一
filtername (fn)		指定语法高亮显示过滤器，用于给定的主模式
for-buffers (fb)	mixed	指定通配（globbing）或正则表达式是否用于for-buffers与kill-buffer命令中以选择缓冲区名称
glob	!echo %s	控制通配符（如*与?）在提示文件名时如何处理。off会取消展开；而on会使用内部方式，可以处理正常的shell通配符与~表示法。Unix的默认值可以保证其与你的shell间的兼容性
highlight (hl)	highlight	在对应的缓冲区中启用或禁用语法高亮显示
history (hi)	history	将冒号命令行（迷你缓冲区）中的命令记录在[History]缓冲区中
horizscroll (hs)	horizscroll	移动到过长行的边缘之外时，将整个屏幕往左右移动。如果没有设置该项，则只有当前的行会移动
ignoresuffix(is)	\(\.orig\\|~\)$	在匹配文件名与主模式后缀前，先从文件名中去除指定的模式
linewrap (lw)	nolinewrap	将逻辑上过长的行绕回，成为屏幕上的多行
maplonger	nomaplonger	映射功能会匹配可能的最长映射序列，而不是最短的
meta-insert-bindings (mib)	mib	控制插入时的8位字符的行为。正常情况下，按键绑定只在命令模式中有用；在插入模式时，所有字符都是自我插入的。如果打开此种模式，输入一个绑定到函数的元字符（即设置第8个位的字符），则此函数的绑定会被接受并在插入模式中执行。任何未绑定的元字符仍然只自我插入
mini-hilite (mh)	reverse	定义用户在迷你缓冲区中切换编辑模式时所使用的高亮显示属性

表B-5：vile 9.6的设置选项（续）

选项	默认值	说明
modeline	nomodeline	控制是否启用类似vi的模式行功能
modelines	5	控制缓冲区各个结尾的行数，以扫描类似vi的模式行
overlap-matches	overlap-matches	调整visual-matches呈现的高亮显示效果，以控制是否显示覆盖的匹配内容
percent-crlf	50	必须以CR/LF结尾的总行数百分比，以供vile自动转换缓冲区的recordseparator为crlf
percent-utf8	90	包含嵌入null的总字符百分比，让它们看起来像UTF-16或UTF-32编码。如果file-encoding选项设置为auto且匹配高于此阈值，vile将以UTF-8载入缓冲区数据
popup-choices (pc)	delayed	控制弹出窗口的使用，以帮助自动补全。其值包括：off表示没有窗口，immediate表示立即弹出窗口，而delayed 表示等待下一个Tab键
popup-msgs (pm)	nopopup-msgs	启用此选项时，vile弹出[Messages]缓冲区，呈现写入消息行的文本
recordseparator (rs)	lfa	指定vile读写的文件格式。格式为lf（用于 Unix）、crlf（用于 DOS）、cr（用于Macintosh）、 default（根据平台决定为lf或crlf）
resolve-links	noresolve-links	如果设置的话，vile会完全解决某些路径中包含符号链接的情况。有助于避免通过不同的路径名称对同一个实际文件做编辑的情况发生
ruler	noruler	在状态行显示当前的行与栏以及光标位置之前的内容占当前缓冲区的百分比
showchar (sc)	noshowchar	于状态行中显示当前字符的值

表B-5：vile 9.6的设置选项（续）

选项	默认值	说明
showformat (sf)	foreign	控制recordseparator信息显示在状态行中本机语言的时间以及是否显示。其值包括：always、differs（在本地模式与全局模式不同时显示）、local（在设置本地模式时显示）、foreign（在 recordseparator与默认本机语言不同时显示）以及never
showmode (smd)	noshowmode	在状态行中显示用于插入与替换模式的指示符
sideways	0	提示左右滚动数量的新值，它会控制屏幕往左或往右滚动的字符数。0值表示移动1/3个屏幕
tabinsert (ti)	tabinsert	允许在缓冲区中实际插入tab字符。如果关闭（notabinsert），vile永远不会在缓冲区中插入tab，而会插入适当数量的空格
tagignorecase (tc)	notagignorecase	让标签搜索忽略大小写
taglength (tl)	0	定义标签中有效字符的数量。默认值（0）表示所有的字符都是有效的。本选项不会影响从光标获得的标签，它们必须永远正确匹配（这与其他编辑器有些不同）
tagrelative (tr)	tagrelative	在其他目录中使用tags文件时，会把 tags文件中的文件名当成相对于tags文件所在的目录而计算
tags	tags	以空格分隔的文件列表，用来查找标签引用
tagword (tw)	notagword	用光标所在位置的整个单词做标签查找，而不是在当前光标位置开始的部分单词
undolimit	10	能够撤销的更改次数限制。设置为0表示“没有限制”
unicode-as-hex (uh)	nounicode-as-hex	如果显示的缓冲区编码为Unicode 之一（例如utf-8、utf-16、utf-32），则非ASCII字符显示为\u*XXXX*格式，即使显示程序能显示这些正规字符
unprintable-as-octal (uo)	nounprintable-as-octal	将第8个位被设置的不可打印字符用八进制显示；否则，使用十六进制。第8个位没有被设置的不可打印字符，永远以控制字符表示法来显示

表B-5：vile 9.6的设置选项（续）

选项	默认值	说明
visual-matches	none	控制所有匹配搜索模式的内容的高亮显示。可能的值包括：none（不做高亮显示），underline（下划线）、bold（黑体）与reverse（高亮）。系统若有支持，也可以使用颜色高亮显示
xterm-fkeys	noxterm-fkeys	通过产生Shift-、Ctrl-、Alt-修饰符（modifier）与每个列在终端描述中的功能键的系统绑定，支持xterm的修改（modified）功能键
xterm-mouse	noxterm-mouse	允许在xterm中使用鼠标。详细内容请参阅在线帮助
xterm-title	noxterm-title	如果在xterm中运行，启用标题栏的更新。每次切换至不同缓冲区时，vile会更新标题。它与xterm-mouse模式使用相同的TERM变量测试

a:　根据编译vile的平台而定。



附录C

问题集

本附录合并了第一部分所提到的问题集以便于查阅。

打开文件时发生的问题

- *调用*vi*时，出现*[open mode]*消息。*

  你的终端类型可能未正确识别。立刻输入:q离开这个编辑会话。检查$TERM环境变量，它应该设置为你的终端名称。也可以查询你的系统管理器，以提供正确的终端类型设置。

- *见到下列任何一种消息：*

  ```
  Visual needs addressable cursor or upline capability
  Bad termcap entry
  Termcap entry too long
  terminal:  Unknown terminal type
  Block device required
  Not a typewriter
  ```

  可能是终端类型没有定义，或是terminfo 或 termcap项中有错误。输入:q离开。检查$TERM环境变量，或要求系统管理器为你的环境选择一个终端类型。

- *当你认为文件已存在时，却出现*[new file]*消息。*

  确定文件名的大小写正确（Unix会区分文件名的大小写）。如果正确，很可能位于错误的目录。输入:q离开。检查你是否位于正确的目录中（在Unix提示符下输入pwd）。如果位于正确的目录中，则检查目录中的文件列表（使用ls），以确定是否有个名称只差一点点的文件。

- *调用*vi*，却得到冒号提示符（表示你在*ex*行编辑模式中）。*

  你可能在vi重绘屏幕前将其中断了。在ex提示符（:）下输入vi 以进入vi。

- *出现以下消息之一：*

```
[Read only]
File is read only
Permission denied
```

"Read only"表示你只能查看文件，而不能保存任何更动。你可能在查看模式（*view mode*，使用view或vi -R）中调用了vi，或者你对文件没有写入的权限。请参阅第443页的"保存文件时发生的问题"一节。

- *出现以下消息之一：*

```
Bad file number
Block special file
Character special file
Directory
Executable
Non-ascii file
file non-ASCII
```

你要调用来编辑的文件不是常规的文本文件。输入:q!离开，再检查你要编辑的文件，可以使用file命令。

- *当你遇到上述困难而输入:q后，却出现如下消息：*

```
No write since last change (:quit! overrides).
```

表示你更改了文件而不自知。输入:q!离开vi。你所做的改变将不会保存到文件中。

保存文件时发生的问题

- *尝试写入文件，却出现以下消息之一：*

```
File exists
File file exists - use w!
[Existing file]
File is read only
```

输入:w! *file*以覆盖现有的文件，或者输入:w *newfile*将编辑的结果写入新的文件。

- *你想写入文件，却没有写入的权限，并得到"Permission denied."消息。*

使用:w *newfile*将缓冲区写入一个新文件。如果你拥有目录的写入权限，则可使用mv将原来的文件用新文件替换。如果你没有目录的写入权限，则输入:w *pathname/file*，将缓冲区写入某个你拥有写入权限的目录（如你的主目录或是/tmp）。

- *尝试写入文件，却得到文件系统已满的消息。*

  输入:!rm *junkfile*来删除一些不需要的（大）文件，空出一些空间（在感叹号后开始ex命令便可以使用Unix）。

  或者输入:!df看看其他文件系统有没有空间。如果有的话，则在其他文件系统中选择一个目录，用:w *pathname*写入你的文件（df是检查磁盘剩余空间的Unix命令）。

- *系统进入开放模式（open mode）并且显示文件系统已满。*

  vi用于临时文件的磁盘已满。输入:!ls/tmp查看有无可以移除的文件，以腾出一些空间（注2）。如果有的话，先创建一个临时的Unix shell，以便移除文件或发出其他Unix命令。你可以输入:sh，创建一个shell；输入CTRL-D或exit，结束shell并回到vi（在现在的Unix系统上，当使用作业控制的shell时，你只需要输入CTRL-Z来暂停vi，即可回到Unix提示符，再输入fg即回到vi）。腾出空间之后，使用:w!写入文件。

- *尝试写入文件，却得到磁盘限额已满的消息。*

  试试以ex命令:pre（:preserve的简写）强迫系统保存你的缓冲区。如果没有用，看看有没有文件可以移除。使用:sh（如果使用的是作业控制系统，也可以用CTRL-Z）暂时离开vi并移除文件。完成之后，使用CTRL-D（或fg）回到vi，再用:w!写入文件。

可视模式的问题

- *在vi中编辑文件时，有时会意外进入ex编辑器。*

  在vi的命令模式中输入Q时会调用ex。若是意外进入ex编辑器，输入命令vi即可回到vi编辑器。

vi 命令的问题

- *输入命令时，文字出乎意料地在屏幕上到处乱跑，一切的运作都不如预期。*

  确定你输入了j，而不是输入了J。

  你可能按了CAPS LOCK键却没有注意到。vi对大小写很敏感，也就是说，大写命令（I、A、J）与小写命令（i、a、j）是不一样的，因此按下CAPS LOCK后输入

注2：vi可能将临时文件放在/usr/tmp、/var/tmp或你的当前目录，你可能需要寻找一下，以检查究竟是哪个空间用完了。

的所有命令都会被当成大写命令。请按CAPS LOCK回到小写状态，再按ESC确定你处于命令模式，然后可输入U以恢复上一行的改变，或者输入u以撤销上一个命令。你可能需要做一些额外的工作，才能恢复刚才弄乱的部分。

删除文件时发生的问题

- *你误删了文本，想要补救。*

  有好几种方法可以恢复被删除的文本。如果你刚刚删除了一些东西，但马上就发觉了，只要输入u就可以撤销上一个命令（如dd）。但这只适用于尚未下达其他命令的时候，因为u只会撤销最近一个命令。另一方面，U会恢复一整行成原来面貌，就是做任何改变之前的样子。

  你也可以使用p命令恢复最近几次的删除动作，因为vi会将最近9次的删除动作保存在9个编号的删除缓冲区中。举个例子，如果你知道要恢复的缓冲区是第三个，则可以输入：

  ```
  "3p
  ```

  把第三个缓冲区“放到”光标所在的下一行上。但这只对被删除的一整行才有用。被删除的单词或是一行的一部分都不会被保存到缓冲区中。如果你要恢复一个单词或一整行的一部分，而且u命令没有用，请单独使用p命令。它会恢复所有你刚刚删除的内容。

附录D

vi与国际互联网

当然，vi是很友善的。只是在选择朋友时很有原则。

至少从1980年开始，vi就是Unix的“标准”满屏编辑器，使它成为Unix文化中的珍宝了。

vi帮忙构建了Unix，而Unix则构建了今日国际互联网的基础。因此，可以预期至少会有一个奉献给vi的网站。本篇附录介绍了一些vi行家所需要的vi资源。

从何开始

在印好的书中介绍万维网上的网站，绝对是很容易过时的事。我们尝试在此列出一些应该会拥有合理寿命的网站。

另外，elvis说明文档中的“Tips”部分，列出了与vi相关的有趣网站（我们就是从此处开始），而Usenet中的comp.editors新闻组也是值得一看的地方。

vi网站

有两个主要的vi相关网站，包括Thomer M. Gil的*vi Lover's Home Page*以及Sven Guckes的*Vi Pages*。它们都包含了许多与vi相关的有趣链接。

vi Lover's Home Page

*vi Lover's Home Page*位于*http://www.thomer.com/thomer/vi/vi.html*。这个站点包括以下项目：

- 已知的vi同类品列表，及其源代码与二进制发行包的链接
- 其他vi站点的链接，包含Sven Guckes的*Vi Pages*
- 大量到vi说明文档、手册、帮助文件与使用说明的链接，分成许多不同的层次

- 用来编写HTML文档以及解“汉诺塔”问题的vi宏，还有提供其他宏的FTP站点
- 各种vi相关链接：诗词、关于vi“真正历史”的故事、vi与Emacs的讨论以及vi咖啡杯（参阅后文的“给Java爱好者的vi”）

也还有其他的东西，这是一个不错的出发点。

Vi Pages

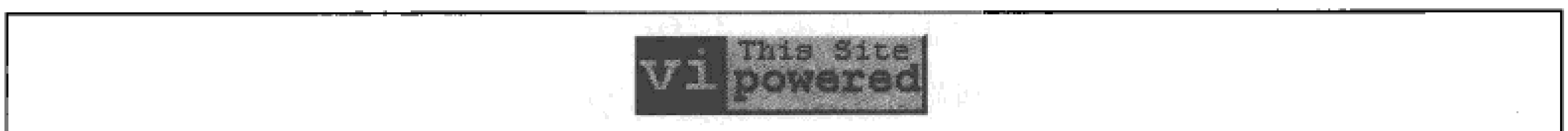
*Vi Pages*位于*http://www.vi-editor.org*（注1）。这个站点包括以下的项目：

- 不同的vi同类品间的选项与特性的详细比较
- vi不同版本的屏幕截图
- 列出许多vi同类品及其作者的联络信息（姓名、地址、URL）的表格
- 指向许多常见问题集（FAQ）文件的链接
- 许多关于vi的格言，像本章开头引用的格言
- 其他的链接，包含指向vi杂货（咖啡杯等）的链接

*vi Lover's Home Page*形容这里为“地球上唯一比你现在浏览的（站点）还要好的Vi站点。”本站也值得好好浏览。

vi Powered!

我们发现一个蛮可爱的*vi Powered*徽标（图D-1）。这是一个小小的GIF文件，你可以将它加在使用vi的自建网页中。



图D-1：vi Powered!

vi Powered! 徽标的原始网页在*http://www.abast.es/~avelle/vi.html*。这个网页以西班牙文设计，已经不复存在。英文版本位于*http://www.darryl.com/vi.shtml*。添加此徽标的说明位于*http://www.darryl.com/addlogo.html*。其中包含了几个简单的步骤：

注1：至中译本出版时，镜像站点共有三处：*http://www.saki.com.au/mirror/vi/index.php3*、*http://mirrors.msfree.ca/vi*、*http://www.leg.uct.ac.za/mirrors/guckes/vi/*。

1. 下载此徽标。在你的（图形化）浏览器中输入*http://www.darryl.com/ vipower.gif*，并将其保存成文件；或使用命令列的网站获取工具程序，例如wget。

2. 在适当的地方加入以下的代码：

   ```
   <A HREF="http://www.darryl.com/vi.html">
   <IMG SRC="vipower.gif">
   </A>
   ```

   上述代码把徽标放入你的网页中并制成超文本链接，选中时会进入*vi Powered*的主页。你可能希望在`<IMG>`标签中加入`ALT="This Web Page is vi Powered"`属性，对使用非图形化浏览器的用户提供方便的参考。

3. 在网页中的`<HEAD>`部分加入以下的代码：

   ```
   <META name="editor" content="/usr/bin/vi">
   ```

就像真正的程序员（Real Programmer）会避开“所见即所得”的文字处理器而使用`troff`，同样地，真正的网站大师（Real Webmaster）也会避开花俏的HTML编辑工具而使用`vi`。你可以用*vi Powered*徽标骄傲地显示这个事实。

你可以在*http://www.vim.org/logos.php*上找到设计不同的Vim徽标。在*http://www.vim.org/buttons.php*上则有许多*Vim Powered*徽标。

给Java爱好者的*vi*

虽然标题如此，但是这里指的是用来喝的爪哇咖啡，不是用来写程序的Java。（注2）

我们假想的“Real Programmer”，使用vi编写C++代码、`troff`文档与网页，无疑地，他偶尔需要一杯咖啡。大家现在可以捧着印有vi命令的咖啡杯来喝咖啡了！

第一次找到具有vi参考资源的杯子时，是由一个专门网站以四个一组的方式提供。该站点似乎消失了，不过，在另一个网站，*http://www.cafepress.com/geekcheat/366808*上，已可买到印了vi参考资源的马克杯、T恤、运动上衣与鼠标垫。

vi的在线使用手册

我们提到的两个网页有许多vi说明文档的链接。特别引人注意的是一份共有9个部分的在线使用手册，出自《Unix World》杂志，作者是Walter Zintz。出发点在*http://www.networkcomputing.com/unixworld/tutorial/009/009.html*（它的链接经常移动，在vi主页上

注2：虽然就某方面而言，也蛮适用于出身于Sun Microsystem的Java，因为Bill Joy——vi的原始作者——是其创办人与副总裁。

的链接不见得有效，但本处列出的URL在我们于2008年测试时仍然有效）。这份使用手册包含了以下主题：

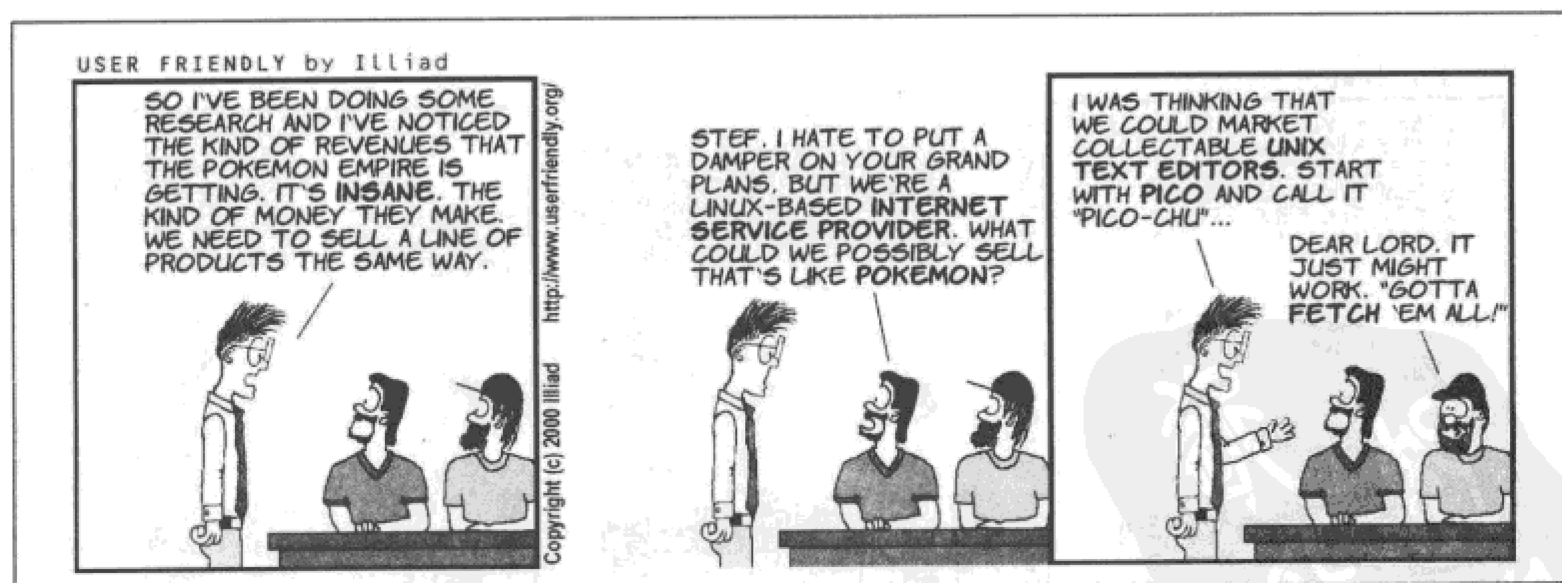
- 编辑器的基础
- 行模式的地址
- g（全局）命令
- 替换命令
- 编辑环境（set命令、标签、EXINIT与.exrc）
- 地址与栏
- 替换命令r与R
- 自动缩排
- 宏

除了使用手册之外，还有一个在线测验，可以让你知道吸收了多少手册中的内容。你也可以直接试试这个测验，看看阅读本书的效果如何！

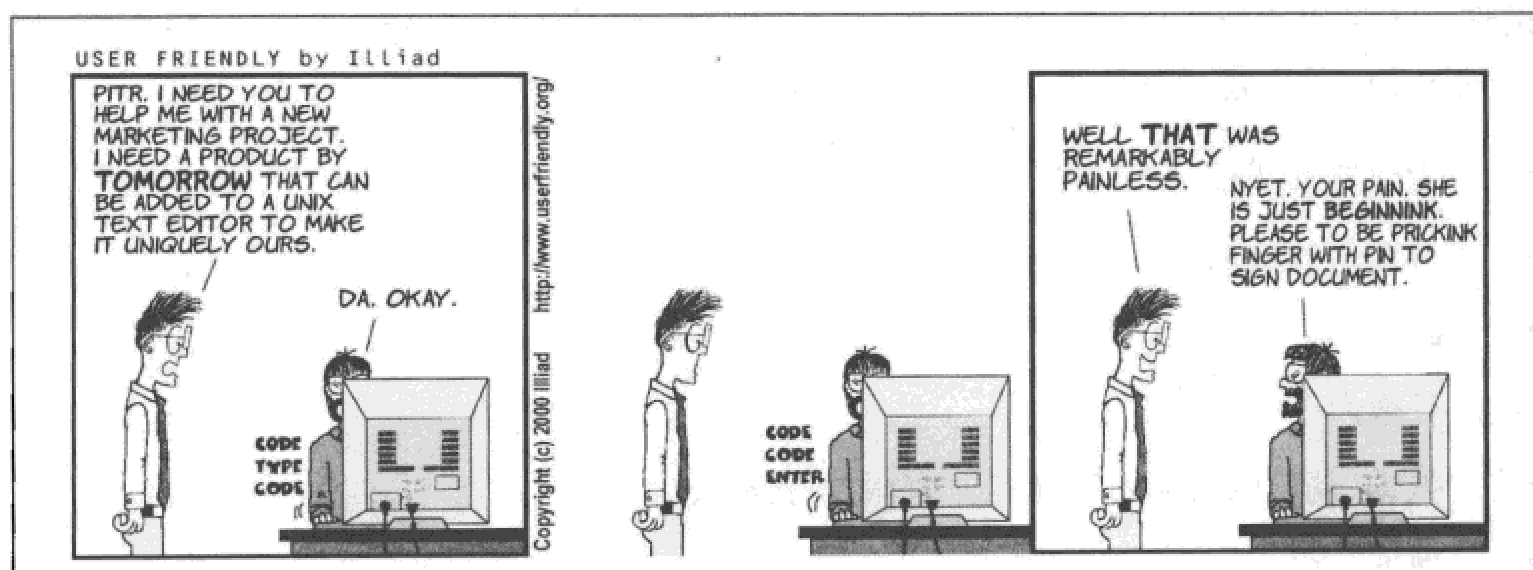
不同的vi同类品

图D-2到图D-9是vigor的故事，它是另一个vi同类品。

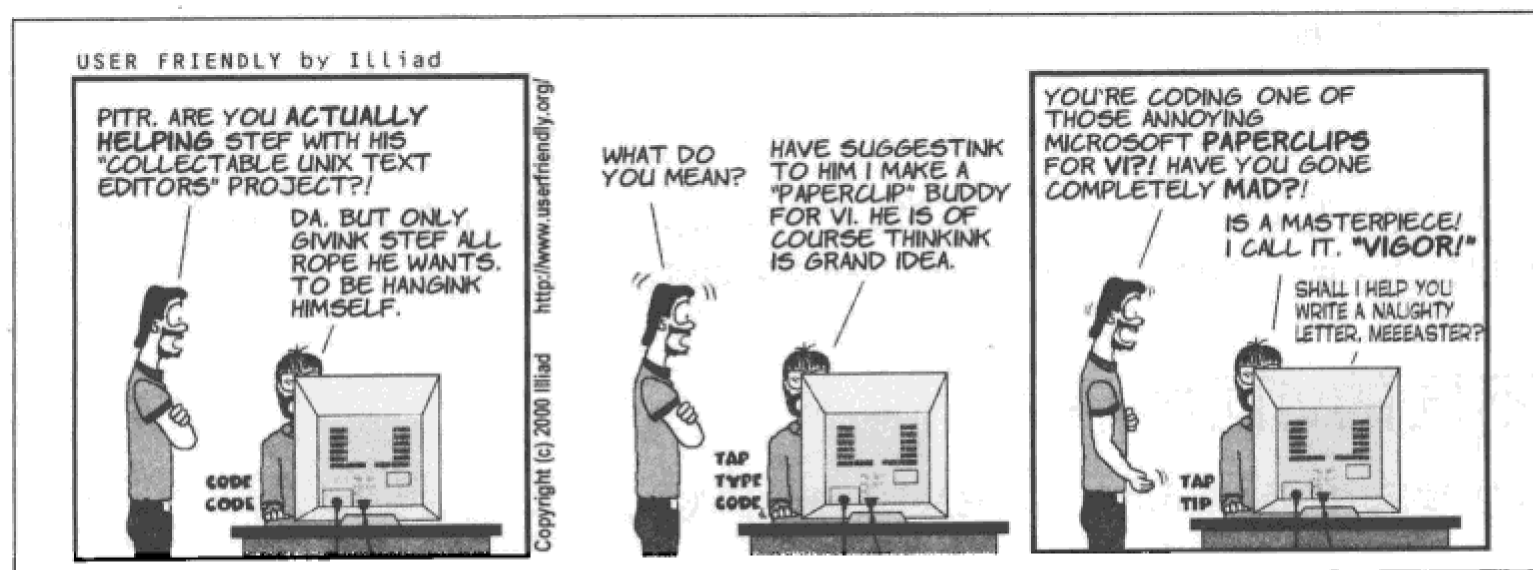
vigor的源代码可至*http://vigor.sourceforge.net*取得。



图D-2：vigor的故事：第1篇



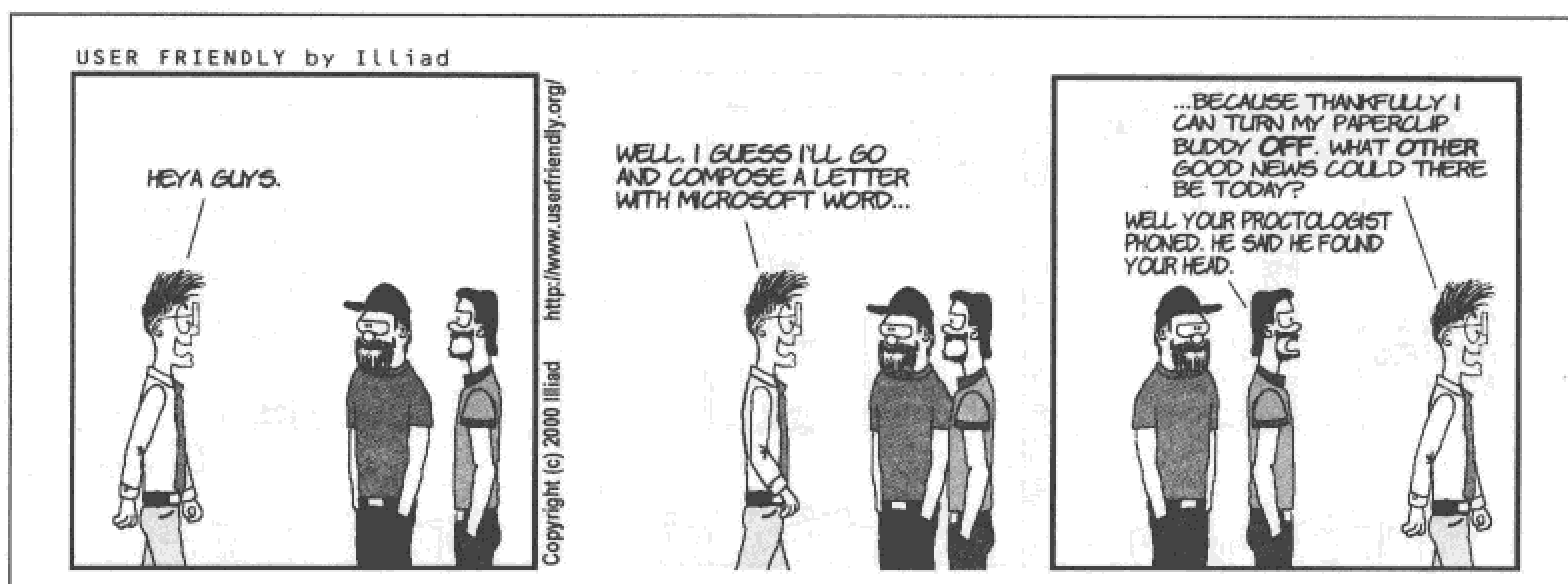
图D-3：vigor的故事：第2篇



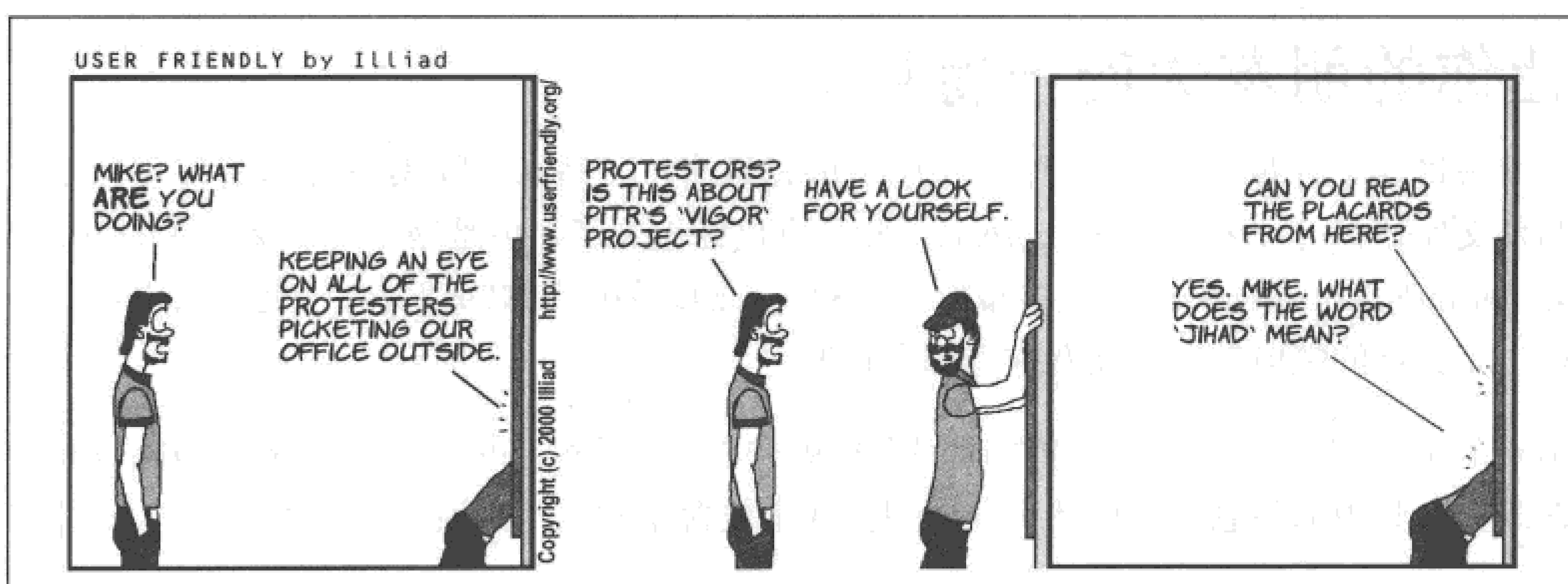
图D-4：vigor的故事：第3篇



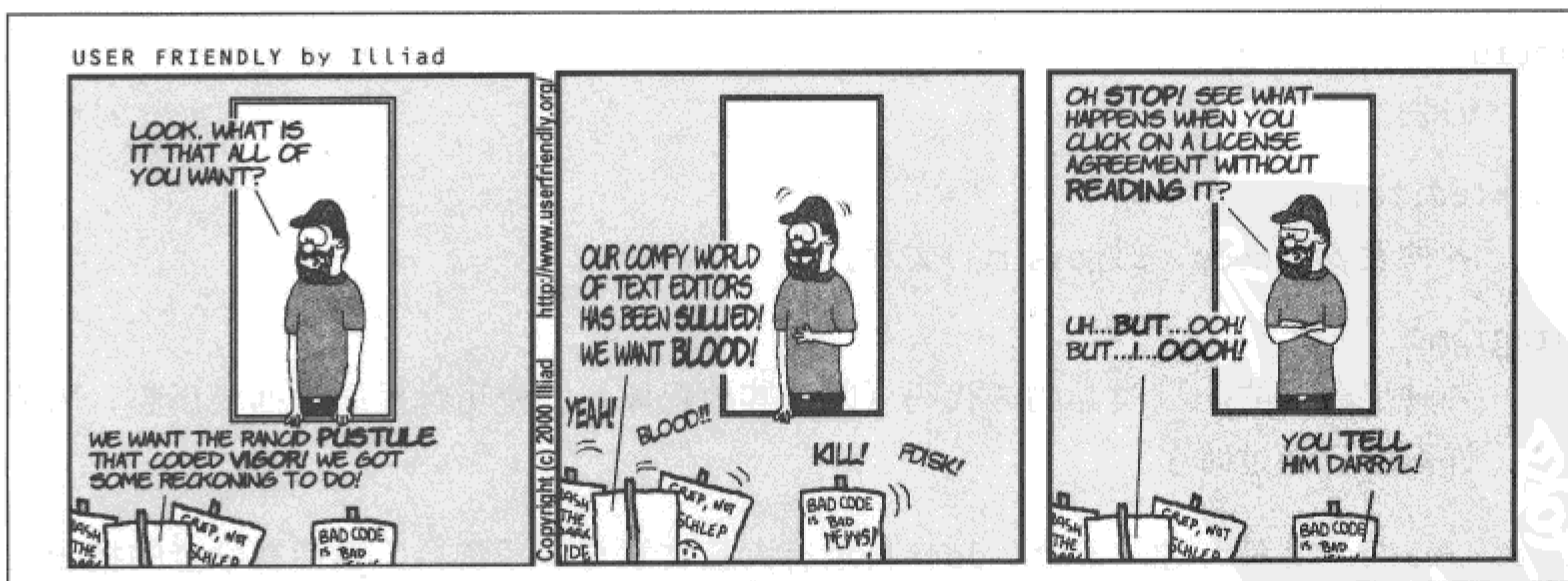
图D-5：vigor的故事：第4篇



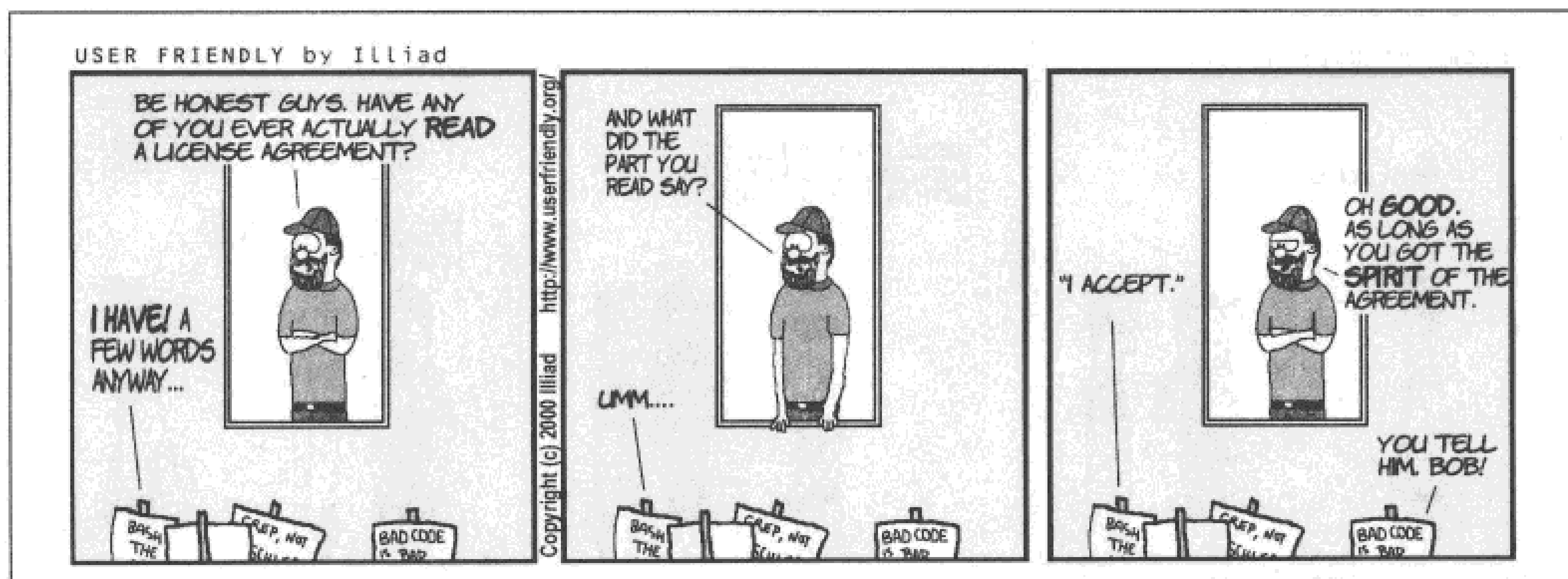
图D-6：vigor的故事：第5篇



图D-7：vigor的故事：第6篇



图D-8：vigor的故事：第7篇



图D-9：vigor的故事：第8篇

让你的朋友大吃一惊吧！

长期来说，也许最有用的项目是在*alf.uib.no*这个FTP站点中收集的vi相关信息。原始归档文件是在*ftp://afl.uib.no/pub/vi*。此站点已经消失了，但是这些归档文件在*ftp://ftp.uu.net/pub/textprocessing/vi*上有镜像（注3）。此目录中的INDEX文件描述了其内容，并行出了其他的镜射站，可能有些离你会较近。

不幸的是，这些文件上一次更新时间在1995年5月。而幸运的是，vi的基本功能没有什么改变，其中的信息与宏仍然可以使用。其中包含了4个子目录：

`docs`
: vi的说明文档以及一些`comp.editors`中的文章。

`macros`
: vi宏。

`comp.editors`
: 各种发表在`comp.editors`上的文章。

`programs`
: 各种平台上的vi同类品的源代码（以及其他程序）。使用这里的数据要小心，因为许多都已经过时了。

`docs`与`macros`是最有趣的部分。`docs`目录有大量文章与参考资料，包括初学者的指导、程序错误的解释、快速参考许多简短的“如何做……”文章（例如，如何在vi中把句子的第一个字母改成大写的）。甚至还有一首关于vi的歌！

注3：使用命令行FTP客户端，可能比使用网站浏览器更容易访问这个站点。

macros目录中有超过50个文件，分别做不同的工作。我们在这里只介绍其中三个。（文件名结尾为.Z的文件是用Unix的compress程序所压缩的。可用uncompress或gunzip解压缩。）

evi.tar.Z
: 一个Emacs“仿真器”。这个点子的背后是将vi转变成一种“无模式的”（modeless）编辑器（永远处在输入模式，而命令由控制键来执行）。它实际上是用一份替换了EXINIT环境变量的shell脚本来达成。

hanoi.Z
: 这可能是最有名的vi特殊应用：一组可以解决“汉诺塔”问题的宏。这个程序只会显示其移动，并不会实际画出圆盘来。为了趣味起见，我们以花絮的形式列出它的内容。

turing.tar.Z
: 这个程序用vi实现出一个真的图灵机！看着程序的执行，真的令人吃惊。

还有许多许多更有趣的宏，包括perl与RCS模式。

vi版的“汉诺塔”宏

```
" From: gregm@otc.otca.oz.au (Greg McFarlane)
" Newsgroups: comp.sources.d,alt.sources,comp.editors
" Subject: VI SOLVES HANOI
" Date: 19 Feb 91 01:32:14 GMT
"
" Submitted-by: gregm@otc.otca.oz.au
" Archive-name: hanoi.vi.macros/part01
"
" Everyone seems to be writing stupid Tower of Hanoi programs.
" Well, here is the stupidest of them all: the hanoi solving
" vi macros.
"
" Save this article, unshar it, and run uudecode on
" hanoi.vi.macros.uu. This will give you the macro file
" hanoi.vi.macros.
" Then run vi (with no file: just type "vi") and type:
" :so hanoi.vi.macros
" g
" and watch it go.
"
" The default height of the tower is 7 but can be easily changed
" by editing the macro file.
"
" The disks aren't actually shown in this version, only numbers
" representing each disk, but I believe it is possible to write
" some macros to show the disks moving about as well. Any takers?
"
```

```
" (For maze solving macros, see alt.sources or comp.editors)
"
" Greg
"
" ------------ REAL FILE STARTS HERE ---------------
set remap
set noterse
set wrapscan
" to set the height of the tower, change the digit in the following
" two lines to the height you want (select from 1 to 9)
map t 7
map! t 7
map L 1G/t^MX/^0^M$P1GJ$An$BGCOe$X0E0F$X/T^M@f^M@h^M$A1GJ@f0l$Xn$PU
map g IL
map I KMYNOQNOSkRTV
map J /^0[^t]*$^M
map X x
map P p
map U L
map A "fyl
map B "hyl
map C "fp
map e "fy2l
map E "hp
map F "hy2l
map K 1Go^[
map M dG
map N yy
map O p
map q tllD
map Y o0123456789Z^[0q
map Q 0iT^[
map R $rn
map S $r$
map T ko0^M0^M^M^[
map V Go/^[
```

好吃又不黏牙

```
vi is [[13~^[[15~^[[15~^[[19~^[[18~^ a[_443_]
muk[^[[29~^[[34~^[[26~^[[32~^ch better editor than this emacs. I know
I^[[14~'ll get flamed for this but the truth has to be
said. ^[[D^[[D^[[D^[[D ^[[D^[^[[D^[[D^[[B^
exit ^X^C quit :x :wq dang it :w:w:w :x ^C^C^Z^D
```

— Jesper Lauridsen, alt.religion.emacs

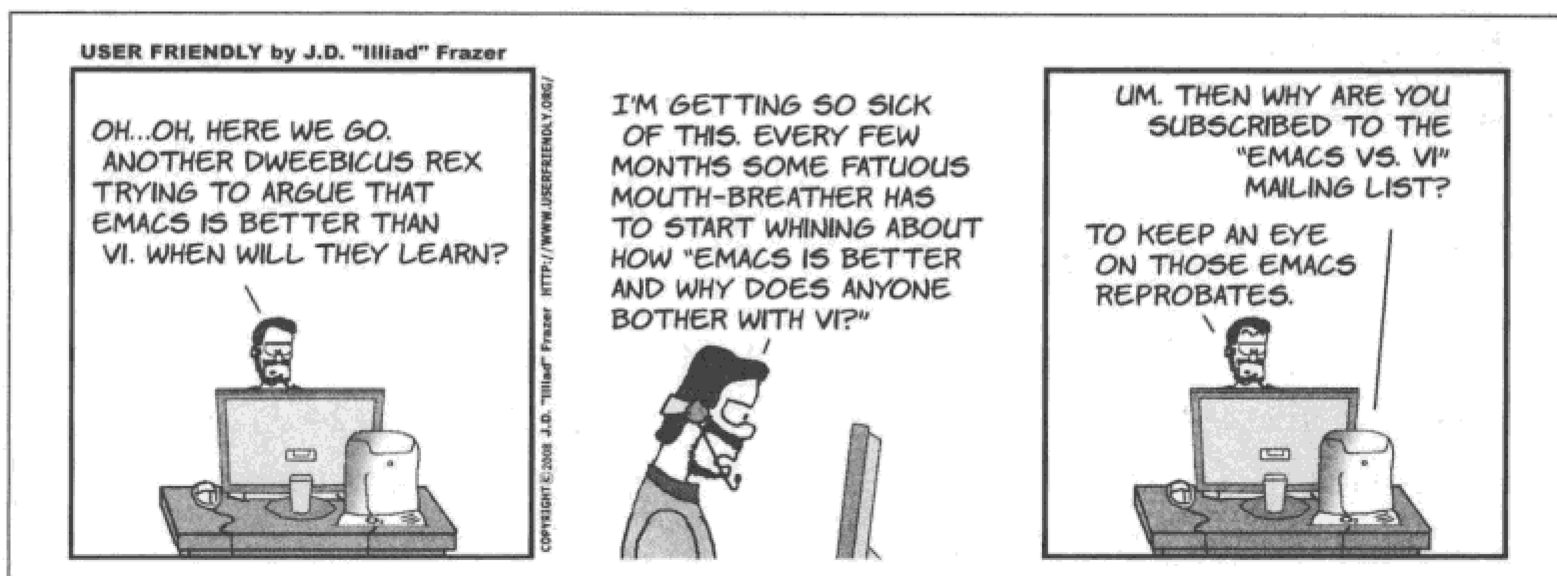
要讨论vi在Unix文化中的地位，就不能不承认在Unix社群（注4）中可能延续最久的争论——vi与Emacs之争。

注4：其实它已经成为宗教战争了，但我们要试着和解一下。（另一场宗教性的战争，BSD与System V之争，因为POSIX而得到解决。System V赢了，不过BSD也得到重大的让步。）

在comp.editors（以及其他新闻组）中，年年都会出现讨论谁好谁坏的文章（图D-10提供了不错的举例说明）。在前面介绍的许多网站中，可以找到一些讨论的整理。也可以找到一些网页当前位置的指示。

一些对vi较有利的论点如下：

- vi在所有Unix系统中都可以使用。如果你在安装系统或转移到别的系统，很可能必须使用vi。



图D-10：它不是一场宗教战争。真的！

- 你通常可以将手指放在键盘上起始行（hom row）上。这对打字熟练的人来说，是一大好处。
- 命令是一个（有时是两个）一般字符，比起Emacs所需的控制字符与元字符，输入时容易多了。
- vi一般比Emacs小并且较省资源。启动时间尤其快，有时可能相差数十倍。
- 现在的vi同类品加入了一些特性，例如增量搜索、多窗口与缓冲区的编辑、图形用户界面、语法高亮显示与智慧缩排，还有通过扩展语言做到的程序化功能等等，使得这两种编辑器功能的差距即使没有消失，也大幅缩小了。

为了完整起见，还要提到两件事。首先，常见的Emacs实际上有两种：原始的GNU Emacs与XEmacs。后者是从早期版本的GNU Emacs发展而来的。它们各有各的优缺点，以及各自的支持者。（注5）

另外，GNU Emacs一直都有仿真vi的软件包，但都不是很好。最近情况不同了。“viper mode”被公认是相当优秀的vi仿真。对于有兴趣的人，它可以作为学习Emacs的桥梁。

注5：但它们的支持者都同样对vi不屑一顾！

总而言之，各位用户才是最后决定程序效用的人。你应该使用能得到最高生产力的工具，而在许多任务中，vi与其同类品都是非常优秀的工具。

vi格言

最后，还有一些vi的格言，由Vim的作者Bram Moolenaar所赞助：

定理：vi是完美的。

证明：VI是罗马数字中的6。可以被6整除又小于6的自然数是1、2、3。1 + 2 + 3 =6。因此6是一个完美数。因此，vi是完美的。

——Arthur Tateishi

Nathan T. Oelger的响应：

> 因此，将上面的结果用在Vim会如何呢？VIM在罗马数字中是代表(1000 - (5 + 1))= 994，等于2*496+2。496可以被1、2、4、8、16、31、62、124 与248整除，而1+2+4+8+16+31+62+124+248 = 496。所以，496是一个完美数。因此，Vim比vi完美两倍，再加上许多的好东西。
>
> 也就是说，Vim比完美更美好。

这一则格言道尽了所有真正的vi爱好者的心声：

> 对我来说，vi就是禅。使用vi，就是参禅。每一个命令都是心印。来自内心深处，非有经验不能明白。每一次使用，都会发现真理。

——Satish Reddy

